Python 2.4 Quick Reference Card

Card Content

Styles : keyword **function/method** type *replaced_expression* **variable** literal **module** module_filename **language_syntax**

Notations :

f(…)→ return value   **f**(…)➤ return nothing (procedure)

[x] for a list of x data, (x) for a tuple of x data, may have x{n}→ n times x data.

Environment Variables

PYTHONCASEOK	[1] no case distinction in module→file mapping
PYTHONDEBUG	[1] = -d command-line option
PYTHONHOME	Modify standard Python libs prefix and exec prefix locations. Use <prefix>[:<execprefix>].
PYTHONINSPECT	[1] = -i command-line option
PYTHONOPTIMIZE	[1] = -O command-line option
PYTHONPATH	Directories where Python search when importing modules/packages. Separator : (posix) or ; (windows). Under windows use registry HKLM\Sofware\….
PYTHONSTARTUP	File to load at begining of interactive sessions.
PYTHONUNBUFFERED	[1] = -u command-line option
PYTHONVERBOSE	[1] = -v command-line option

[1] If set to non-empty value.

Command-line Options

python [-dEhiOQStuUvVWx] [-c *cmd* | -m *mod* | *file* | -] [*args*]

-d	Output debugging infos from parser.
-E	Ignore environment variables.
-h	Print help and exit.
-i	Force interactive mode with prompt (even after script execution).
-O	Optimize generated bytecode, remove assert checks.
-OO	As -O and remove documentation strings.
-Q *arg*	Division option, *arg* in [old(default),warn,warnall,new].
-S	Don't import site.py definitions module.
-t	Warn inconsistent tab/space usage (-tt exit with error).
-u	Use unbuffered binary output for stdout and stderr.
-U	Force use of unicode literals for strings.
-v	Trace imports.
-V	Print version number and exit.
-W *arg*	Emit warning for *arg* "action:message:category:module:lineno"
-x	Skip first line of source (fort non-Unix forms of #!cmd).
-c *cmd*	Execute *cmd*.
-m *mod*	Search module *mod* in sys.path and runs it as main script.
file	Python script file to execute.
args	Command-line arguments for *cmd*/*file*, available in **sys.argv[1:]**.

Files Extensions

.py=source, .pyc=bytecode, .pyo=bytecode optimized, .pyd=binary module, .dll/.so=dynamic library.

.pyw=source associated to pythonw.exe on Windows platform, to run without opening a console.

Language Keywords

List of keywords in standard module **keyword**.

and as[1] assert break class continue def del elif else except exec finally for from global if import in is lambda not or pass print raise return try while yield

[1] not reserved, but avoid to redefine it.

Don't redefine these constants : None, True, False.

Builtins

Available directly everywhere with no specific import. Defined also in module **__builtins__**.

Types

basestring[1] bool buffer complex dict exception file float frozenset int list long object set slice str tuple type unicode xrange

[1] basestring is virtual superclass of str and unicode.
This doc uses string when unicode and str can apply.

Functions

Constructor functions of builtin types are directly accessible in builtins.

__import__ abs apply[1] callable chr classmethod cmp coerce compile delattr dir divmod enumerate eval execfile filter getattr globals hasattr hash help hex id input intern[2] isinstance issubclass iter len locals map max min oct open ord pow property range raw_input reduce reload repr reversed round setattr sorted staticmethod sum super unichr vars zip

[1] Use f(*args,**kargs) in place of apply(f,args,kargs).
[2] Don't use intern.

Statements

One statement per line[1]. Can continue on next line if an expression or a string is not finished (([{ """ ''' not closed), or with a \ at end of line.
Char # start comments up to end of line.

pass	Null statement.
assert *expr*[,*message*]	Assertion check expression true.
del *name*[,…]	Remove name → object binding.
print [>>*obj*,][*expr*[,…][,]	Write *expr* to **sys.stdout**[2].
exec *expr* [in *globals* [, *locals*]]	Execute *expr* in namespaces.
fct([*expr*[,…]], [*name*=*expr*[,…]] [,**args*][,***kwargs*])	Call any callable object *fct* with given arguments (see Functions Definitions & Usage - p2).
name[,…] = *expr*	Assignment operator[3].

[1] Multiple statements on same line using ; separator - avoid if not necessary.
[2] Write to any specified object following file interface (write method).
Write space between expressions, line-return at end of line except with a final ,.
[3] Left part name can be container expression. If expr is a sequence of multiple values, can unpack into multiple names. Can have multiple assignments of same value on same line : a = b = c = expr.

Other statements (loops, conditions…) introduced in respective parts.

Blocks

A : between statements defines dependant statements, written on same line or written on following line(s) with deeper indentation.
Blocks of statements are simply lines at same indentation level.

```
if x<=0 : return 1
if asin(v)>pi/4 :
    a = pi/2
    b = -pi/2
else :
    a = asin(v)
    b = pi/2-a
```

Statement continuation lines don't care indentation.
To avoid problems, configure your editor to use 4 spaces in place of tabs.

Assignment Shortcuts

a += *b*	*a* -= *b*	*a* *= *b*	*a* /= *b*	
a //= *b*	*a* %= *b*	*a* **= *b*		
a &= *b*	*a* |= *b*	*a* ^= *b*	*a* >>= *b*	*a* <<= *b*

Evaluate *a* once, and assign to *a* the result of operator before =

applied to current *a* and *b*. Example : a%=b ≈ a=a%b

Console & Interactive Input/Output

print *expression*[,…]
input([*prompt*]) → evaluation of user input (typed data)
raw_input([*prompt*]) → str: user input as a raw string
Direct manipulation (redefinition) of stdin/stdout/stderr via sys module :
sys.**stdin** **sys**.**stdout** **sys**.**stderr**
sys.**__stdin__** **sys**.**__stdout__** **sys**.**__stderr__**
All are files or files-like objects. The **__xxx__** forms keep access to original standard IO streams.
Ctrl-C raises KeyboardInterrupt exception.
_ → value of last expression evaluation
help([*object*]) ➤ print online documentation
sys.**displayhook** → (rw) fct(value) called to display value
sys.**__displayhook__** → backup of original displayhook function
sys.**ps1** → str: primary interpreter prompt
sys.**ps2** → str: secondary (continuation) interpreter prompt

See external package **ipython** for an enhanced interactive Python shell.

Objects, Names and Namespaces

Identifiers

Use : [a-zA-Z_][a-zA-Z0-9_]*
Special usage for underscore :
_xxx global not imported by import *
_xxx implementation detail, for internal use (good practice)
__xxx 'private' class members, defined as _ClassName__xxx
__xxx__ normally reserved by Python
Case is significant : This_Name != THIS_NAME.

Objects and Names, Reference Counting

Data are typed objects (all data), names are dynamically bound to objects.
= assignment statement bind result of right part evaluation into left part name(s)/container(s). Examples :

```
a = 3*c+5    a,b = ("Hello","World")  x,y,tabz[i] = fct(i)
s = "Hello"  pi,e = 3.14,2.71         a,b = b,a
```

When an object is no longer referenced (by names or by containers), it is destroyed (its **__del__** method is then called).

sys.**getrefcount**(*object*)→ int: current reference counter of *object*

Standard module **weakref** define tools to allow objects to be garbage collected when necessary and dynamically re-created on-demand.

Mutable/Immutable Objects

Mutable objects can be modified in place. Immutable objects cannot be modified (must build a new object with new value).
Immutable : bool, int, long, float, complex, string, unicode, tuple, frozenset, buffer, slice.
Mutable : list, set, dict and other high level class objects.
There is no constant definition. Just use uppercase names to identify symbols which must not be modified.

Namespaces

Places where Python found names.
Builtins namespace → names from module **__builtins__**, already available.
Global namespace → names defined at module level (zero indentation).
Local namespace → names defined in methods/functions.

del *name* ➤ remove existing name from namespace (remove object binding)

globals() → dict: identifier→value of global namespace
locals() → dict: identifier→value of local namespace

Current scope → names directly usable. Searched in locals, then locals from enclosing definitions, then globals, then builtins.
Out-of-scope name → use the dotted attribute notation **x**.**y** (maybe **x**.**y**.**z**.**t**)… where **x** is a name visible within the current scope.

Class namespace → names defined in a class (class members).
Object namespace → names usable with *object*.*name* notation (attributes, methods).

Namespaces can be nested, inner namespaces hidding identical names from outer namespaces.
dir([*object*]) → list: names defined in object namespace[1]
vars([*object*]) → dict[2]: identifier:value of object as a namespace[1]
[1] if object not specified use nearest namespace (locals).
[2] must not be modified.

Constants, Enumerations

Use uppercase and _ for constants identifiers (good practice). May define namespaces to group constants. Cannot avoid global/local name redefinition (can eventually define namespaces as classes with attributes access control - not in Python spirit, and execution cost).
See third party modules **pyenum** for strict enum-like namespace.

Flow Control

Condition

if *cond* : *inst*
[elif *cond* : *inst*]
[else : *inst*]

There is no 'switch' or 'case'.
Can use if elif elif... else.
Can use a mapping with functions bound to cases.

Loop

for *var*[,…] in *iterable* : *inst*
[else : *inst*]

while *cond* : *inst*
[else : *inst*]

Exit loop with break.
Go to next iteration with continue.

Loops else blocs only executed when loop exit normally (without break).

Functions/methods exit

Exit function/method with return [*value*]
Exit from generator body with yield *value*
Multiple returned values using tuple data.
Cannot yield within a try/finally block.

Exceptions

try : *inst*
except [*except_class* [,*value*]] : *inst*
…
[else : *inst*]
Can have a tuple of classes for except_class. Not specifying a class catch all exceptions.
Block else executed when try block exit normally.

try : *inst*
finally : *inst*

Process finally block in all execution paths (normal or exception).
raise *exception_class*[,*value*[,*traceback*]]
raise *exception_object*
raise
Last form re-raise the currently catched exception in an exception handler.

Iterable Protocol

Generic and simple protocol allowing to iterate on any collection of data.
Objects of class defining **__iter__** or **__getitem__** are iterable (directly usable in for loops).
__iter__(**self**) → iterator on **self**
iter(*object*) → iterator on iterable *object*

iter(*callable*,*sentinel*) → iterator returning callable() values up to sentinel
enumerate(*iterable*)→ iterator returning tuples (index,value) from iterable

Iterators Objects Interface

next(**self**)→ next item[1]
__iter__(**self**)→ iterator object itself
[1] When reach end of collection, raise StopIteration exception on subsequent calls (ie. iterator usable only one time on a collection).

Generators

Functions retaining their state between two calls. Return values using yield. Stop generation via simple return or via raise StopIteration.
1) build generator from function : *gen*=*generatorfct*(*args*)
2) use *gen*.**next**() values until StopIteration is raised.

Generator iterable expressions with : (*x* for *x* in *iterable* where *cond*)

Operations with/on Iterable

See Operations on Containers (p8).
See Iteration Tools (p9).

Interpretation / Execution

compile(*string*[1],*filename*,*kind*[2][,*flags*[3][,*dont_inherit*[3]]]) → code object
eval(*expression*[,*globals*[,*locals*]]) → value: evaluation[4] of *expression* string
eval(*code_object*[,*globals*[,*locals*]]) → value: evaluation[4] of *code_object*
exec[5] *statements* [in *globals*[,*locals*]] ➤ *statements* string[1] executed[4]
execfile(*filename*[,*globals*[,*locals*]]) ➤ file *filename* interpreted[4]
[1] Multi-line statements in source code must use \n as newline, and must be terminated by a newline.
[2] Kind relative to string content, 'exec' → sequence of statements, 'eval' → single expression, 'single' → single interactive statement.
[3] Flags and dont_inherit are for future statements (see doc).
[4] In context of globals and locals namespaces.
[5] Exec is a langage statement, others are builtin functions.

Functions Definitions & Usage

def *fctname*([*paramname*[=*defaultvalue*][,…]] [,**args*][,***kwargs*]) :
instructions
new.**function**(*code*,*globals*[,*name*[,*argdefs*]]) → python function (see docs)

Parameters / Return value

Parameters are passed by references to objects.
You can modify values of mutable objects types.
You cannot modify values of immutable objects types - as if they were passed by value.
Notation * → variable list of anonymous parameters in a tuple.
Notation ** → variable list of named parameters in a dict.
Return value(s) with return [*value*[,…]]
For multiple values, return a tuple. If no return value specified or if end of function definition reached, return None value.

Lambda functions

lambda *param*[,…] : *expression*
Anonymous functions defined inline. Result of *expression* evaluation is returned (it must be an expression, no loop, no condition).
Expression uses values known at definition time (except for params).

Callable Objects

Objects having a **__call__** method can be used as functions.
Methods bound to objects can be used as functions : *f* = *o*.*meth*
callable(*x*) → bool: test *x* callable with *x*(…)

Calling Functions

[*name*=] *fctname*([*expr*[,...]][,*name*=*expr*[,...][,**args*][,***args*])

Anonymous parameters passed in parameters order declaration.
Params having default value can be omitted.
Notation * → pass variable list of anonymous parameters in a tuple.
Notation ** → pass variable list of named parameters in a dict.

Functions Control

`sys.getrecursionlimit()`→ int: current recursion limit for functions
`sys.setrecursionlimit(limit)` ➤ set recursion limit for functions

Decorators

Glue code (functions) called at functions and methods definitions time, return the final function/method (generally with wrapping code).

```
@decoratorname [(decorator_arguments)] [...]
def fct(fct_rguments):...
```

`@dec1 @dec2(args) @dec3` `def fct(...):...`	like ➤	`def fct(...):...` `fct = dec1(dec2(args)(dec3(fct))))`

See page PythonDecoratorLibrary in python.org Wiki for some decorators definitions.

Types/Classes & Objects

All data are typed objects relying to classes.
`type(o)` → type: type object of *o*
Standard module **types** define type objects for builtins types.

Class Definition

```
class classname [(parentclass[,...])] :
    varname = expr  ➤ varname defined in classname namespace
    def metname(self[,...]):  ➤ define methods like functions
```

Support multiple inheritance. Can inherit from builtin class.
Inherit at least from object base class => Python 'new style class'.
First parameter of methods is target object, standard use self name.
Access class members via class name, object members via self.
This doc consider you use new style class (inheriting from object).
`new.classobj(name,baseclasses,dict)` → new class (see docs)
`new.instancemethod(fct,instance,class)` → new method: bound to *instance* it it is not None, see docs

Metaclass

Class definition create a new type. It can be done 'by hand' with :

```
x = type('classname',(parentclass,[...]),{varname:expr[,...]}
def metname(self[,...]):
x.metname = metname
```

This allow creation of metaclass class (class building other class).

Object Creation

obj = *ClassName*(*initargs*...)

In case of exception during initialization, object is destroyed when exiting init code (reference counter reach zero).

`new.instance(class[,dict])` → object: create new *class* instance without calling `__init__` method, *dict* is initial object attributes

Classes & Objects Relations

`isinstance(obj,classinfo)` → bool: test object kind of type/class classinfo
`issubclass(aclass,aparent)` → bool: test same class or parent relationship

Prefer isinstance() to type() for type checking.

Parent class methods are not automatically called if overriden in subclass - they must be explicitly called if necessary.
Call parent methods via **super** function :

```
super(ThisClass,self).methodname(self,args...)
```

Or the old way, via parent class namespace :

```
ParentClass.methodname(self,args...)
```


Attributes Manipulation

object.*name* = *value*
`setattr(object,name,value)` ➤ object attribute set to value
object.*name* → *value of object attribute*
`getattr(object,name[,default])`→ value of object attribute
`del object.name`
`delattr(object,name)` ➤ object attribute removed

Special Methods

Other special overridable `__xxx__` methods are listed in respective sections.

Object Life

`__new__(classref,initargs...)`→ object of classref type, already initialized[1]
`__init__ (self,initargs...)`➤ called to initialize object with initargs
`__del__ (self)`➤ called when object will be destroyed

[1] If don't return a classref object, then object.`__init__` is called with initargs.

Object Cast

`__repr__(self)`→ str: called for `repr(self)` and `` `self` ``
`__str__(self)` → str: called for `str(self)` and `print self`
`__coerce__(self,other)` → value, called for `coerce(self,other)`

Object Hash Key

`__hash__(self)`→ int: 32 bits hash code for object, used for `hash(obj)` and quick dict mapping keys comparison - default implementation use `hash(id(self))`

Attributes access

See also "Descriptors protocol" infra.

`__getattr__(self,name)`→ value, called for undefined attributes
`__getattribute__(self, name)`→ value, always called
`__setattr__(self, name, value)` ➤ called for *obj*.*name*=*value*
`__delattr__(self, name)` ➤ called for `del obj.name`
`__call__(self, *args, **kwargs)`→ value, called for *obj*(...)

Static method / Class method

Use standard decorators (see Decorators p3).

```
class ClassName :
    @staticmethod
    def methodname(...): ...
    @classmethod
    def methodname(classref,...): ...
```

Descriptors protocol

Descriptors are attribute objects controling access to attributes values. They must define some of following methods :

`__get__(self,obj,ownerclass)`→ attribute value for *obj*
`__set__(self,obj,value)` ➤ modify attribute in *obj*, set to *value*
`__delete__(self,obj)` ➤ remove attribute from *obj*

In these methods **self** is the descriptor object, and *obj* is the target object which attribute is manipulated.

Properties

A descriptor to directly bind methods/functions to control attribute access. Use builtin type property with init args.

```
class MyClass :
    attributename = property(getter,setter,deleter,description)
```

Each init arg default to None (ie. undefined).

Copying Objects

Assignment only duplicate references. To shallow copy an object (build a new one with same values - referencing same content), or to deep copy an object (deep-copying referenced content), see object copy methods, and functions in standard module **copy**.

`copy.copy(object)`→ value: shallow copy of object
`copy.deepcopy(object[[,memo],_nil])`→ value: deep copy of object[1]

[1] Params **memo** and **nil** are used in recursive deepcopy, their default values are None and empty list.

Copy Protocol

`__copy__(self)`→ value: shallow copy of self, called by `copy.copy(...)`
`__deepcopy__(self,memo)`→ value: deep copy of self, called by `copy.deepcopy(...)`

For copying, objects can define pickling protocol too (see Files - Serialization - p12), in place of `__copy__` and `__deepcopy__`.

Introspection

Beyond this documentation. Many `__xxx__` attributes are defined, some are writable (see other docs).
See standard module **inspect** to manipulate these data.

Example of Introspection Attributes

Note: classes are objects too!

`__base__` → list: parent classes of a class
`__slots__` → tuple: allowed objects attributes names1 of a class
`__class__` → class/type: object's class
`__dict__` → dict: defined attributes (object namespace) of an instance
`__doc__` → string: documentation string of a package, module, class, function
`__name__` → str: object definition name of a function
`__file__` → string: pathname of loaded module .pyc, .pyo or .pyd

[1] List of allowed attributes names. Usage discouraged.

Modules and Packages

File `gabuzo.py` ➤ module **gabuzo**.
Directory `kramed/` with a file `__init__.py` ➤ package **kramed**.
Can have sub-packages (subdirectories having `__init__.py` file).
Searched in the **Python PATH**.
Current Python PATH stored in **sys.path** list. Contains directories and .zip files paths. Built from location of standard Python modules, PYTHONPATH environment variable, directory of main module given on command line, data specified in lines of *.pth* files found in Python home directory, and data specified in registry under Windows.
Current list of loaded modules stored in **sys.modules** map (main module is under key `__main__`).

```
import module [as alias] [,...]
from module import name [as alias] [,...]
from module import *
```

`reload(module)` ➤ *module* is reloaded (but existing references still refer old *module* content)
`new.module(name[,doc])` → new module object.

Import can use package path (ex:`from encoding.aliases import...`).
Direct import from a package use definitions from `__init__.py` file.
Very careful with `import *` as imported names override names already defined.
To limit your modules names exported and visible by `import *`, define module global `__all__` with list of exported names (or use global names `_xxx`).

See `__import__` builtin function, and modules **imp**, **ihooks**.
`__import__(modulename[, globals[,locals[,lnameslist]]])`

Source encodings

See PEP 263. Declare source files encoding in first or second line in a special comment.

```
# -*- coding: encoding_name -*-
```

If this is not specified, Python use `sys.getdefaultencoding()` value (see modules `sitecustomize.py` and `user.py`).

It is important to specify encoding of your modules as u"..." strings use it to correctly build unicode literals.

Special Attributes

`__name__` → str: module name, `'__main__'` for command-line called script

`__file__` → string: pathname of compiled module loaded

Main Execution / Script Parameters

The 'main' module is the module called via command-line (or executed by shell with first script line `#! /bin/env python`).
Command-line parameters are available in `sys.argv` (a python `list`).
At end of module, we may have :

```
if __name__=='__main__' :
    # main code
    # generally call a 'main' function:
    mainfunction(sys.argv[1:])
    # or in lib modules, execute test/demo code...
```

Execution exit after last main module instruction (in multithread, wait also for end of non-daemon threads), unless interactive mode is forced.
Can force exit with calling `sys.exit(`*code*`)`, which raise a `SystemExit` exception - see Current Process - Exiting (p13).

Operators

Deal with arithmetic, boolean logic, bit level, indexing and slicing.

Priority

1	`(a,…)` `[a,…]` `{a:b,…}` `` `…` ``	6	`x+y` `x-y`	11	`x<y` `x<=y` `x>y` `x>=y` `x==y` `x!=y` `x<>y` `x is y` `x is not y` `x in s` `x not in s`
2	`s[i]` `s[i:j]` `s.attr` `f(…)`	7	`x<<y` `x>>y`	12	`not x`
3	`+x` `-x` `~x`	8	`x&y`	13	`x and y`
4	`x**y`	9	`x^y`	14	`x or y`
5	`x*y` `x/y` `x%y`	10	`x|y`	15	`lambda args:expr`

Arithmetic Operators

Can be defined for any data type.

Arithmetic Overriding

`__add__(self,other)` → value: called for `self + other`
`__sub__(self,other)` → value: called for `self - other`
`__mul__(self,other)` → value: called for `self * other`
`__div__(self,other)` → value: called[1] for `self / other`
`__truediv__(self,other)` → value: called[2] for `self / other`
`__floordiv__(self,other)` → value: called for `self // other`
`__mod__(self,other)` → value: called for `self % other`
`__divmod__(self,other)` → value: called for `divmod(self,other)`
`__pow__(self,other)` → value: called for `self ** other`
`__nonzero__(self)` → value: called for `nonzero(self)`
`__neg__(self)` → value: called for `-self`
`__pos__(self)` → value: called for `+self`
`__abs__(self)` → value: called for `abs(self)`
`__iadd__(self,other)` ➤ called for `self += other`
`__isub__(self,other)` ➤ called for `self -= other`
`__imul__(self,other)` ➤ called for `self *= other`
`__idiv__(self,other)` ➤ called[1] for `self /= other`
`__itruediv__(self,other)` ➤ called[2] for `self /= other`
`__ifloordiv__(self, other)` ➤ called for `self //= other`
`__imod__(self,other)` ➤ called for `self %= other`
`__ipow__(self,other)` ➤ called for `self **= other`

[1] without / [2] with from __futur__ import division
Binary operators __xxx__ have also __rxxx__ forms, called when target object is on right side.

Comparison Operators

Operators can compare any data types.
Compare **values** with `<` `<=` `>` `>=` `==` `!=` `<>`.
Test objects **identity** with `is` and `is not` (compare on `id(`*obj*`)`).
Direct composition of comparators is allowed in expressions : x<y<=z>t.
Builtin function `cmp(o1,o2)` → `-1` (o1 < o2), `0` (o1 == o2), `1` (o1 > o2)

Comparison Overriding

`__lt__(self, other)` → `bool`[1]: called for `self < other`
`__le__(self, other)` → `bool`[1]: called for `self <= other`
`__gt__(self, other)` → `bool`[1]: called for `self > other`
`__ge__(self, other)` → `bool`[1]: called for `self >= other`
`__eq__(self, other)` → `bool`[1]: called for `self == other`
`__ne__(self, other)` → `bool`[1]: called for `self != other` and for `self <> other`
`__cmp__(self,other)` → `int`: called for `self` compared to other, `self` <other→value<0, `self`==other→value=0, `self`>other→value>0

[1] Any value usable as boolean value, or a `NotImplemented` value if cannot compare with such other type.

Operators as Functions

Operators are also defined as functions in standard `operator` module.

Comparison

`lt(a,b) = __lt__(a,b)`
`le(a,b) = __le__(a,b)`
`eq(a,b) = __eq__(a,b)`
`ne(a,b) = __ne__(a,b)`
`ge(a,b) = __ge__(a,b)`
`gt(a,b) = __gt__(a,b)`

Logical / Boolean

`not_(o) = __not__(o)`
`truth(o)`
`is_(a,b)`
`is_not(a,b)`
`and_(a,b) = __and__(a,b)`
`or_(a,b) = __or__(a,b)`
`xor(a,b) = __xor__(a,b)`

Arithmetic

`abs(o) = __abs__(o)`
`add(a,b) = __add__(a,b)`
`sub(a,b) = __sub__(a,b)`
`mul(a,b) = __mul__(a,b)`
`div(a,b) = __div__(a,b)`
`mod(a,b) = __mod__(a,b)`
`truediv(a,b) = __truediv__(a,b)`
`floordiv(a,b) = __floordiv__(a,b)`
`neg(o) = __neg__(o)`
`pos(o) = __pos__(o)`
`pow(a,b) = __pow__(a,b)`

Bit Level

`lshift(a,b) = __lshift__(a,b)`
`rshift(a,b) = __rshift__(a,b)`
`inv(o) = invert(o) = __inv__(o) = __invert__(o)`

Sequences

`concat(a,b) = __concat__(a,b)`
`contains(a,b) = __contains__(a,b)`
`countOf(a,b)`
`indexOf(a,b)`
`repeat(a,b) = __repeat__(a,b)`
`setitem(a,b,c) = __setitem__(a,b,c)`
`getitem(a,b) = __getitem__(a,b)`
`delitem(a,b) = __delitem__(a,b)`
`setslice(a,b,c,v) = __setslice__(a,b,c,v)`
`getslice(a,b,c) = __getslice__(a,b,c)`
`delslice(a,b,c) = __delslice__(a,b,c)`

Type Testing

These functions must be considered as not reliable.
`isMappingType(o)`
`isNumberType(o)`
`isSequenceType(o)`

Attribute and Item Lookup

`attrgetter(`*attr*`)` → fct: where fct(x)→x.*attr*
`itemgetter(`*item*`)` → fct: where fct(x)→x[*item*]

Booleans

False : `None`, zero numbers, empty containers. `False` → `0`.
True : if not false. `True` → `1`.
`bool(`*expr*`)` → `True` | `False`
Logical not : `not` *expr*
Logical and : *expr1* `and` *expr2*
Logical or : *expr1* `or` *expr2*
Logical `and` and `or` use short path evaluation.

Bool Cast Overriding

`__nonzero__(self)` → `bool`: test object itself[1]
[1] If __nonzero__ undefined, look at __len__, else object is true.

Numbers

Builtin integer types : `int` (like C long), `long` (unlimited integer)
`int(`*expr*`[,`*base=10*`])` → `int`: cast of *expr*
`long(`*expr*`[,`*base=10*`])` → `long`: cast of *expr*
Builtin floating point types : `float` (like C double), `complex` (real and imaginary parts are `float`).
`float(`*expr*`)` → `float`: *representation of expr*
`complex(`*x*`[,`*y*`])` → `complex`: *number: x+yj*
`[`*x*`+]`*y*`j` → `complex`: *number, ex: 3+4j -8.2j*
c`.real` → `float`: *real part of complex number*
c`.img` → `float`: *imaginary part of complex number*
c`.conjugate()` → `complex`: *conjugate of complex number (real,-img)*

Maximum int integer in sys.maxint.
Automatic conversions between numeric types.
Automatic conversions from int to long when result overflow max int.
Direct conversions from/to strings from/to int, long… via types constructors.

Type `Decimal` defined in standard module `decimal`.
Base fixed type compact storage arrays in standard module `array`.

Operators

`-x` `+x` `x-y` `x*y` `x/y`[1] `x//y`[1] `x%y`[2] `x**y`[2]
[1] With `from __future__ import division`, `/` is true division (`1/2→0.5`), and `//` is floor division (`1//2→0`). Else for integers `/` is still floor division.
[2] `%` is remainder operator, `**` is power elevation operator (same as `pow`).

Functions

Some functions in builtins.
`abs(`*x*`)` → absolute value of *x*
`divmod(`*x*`,`*y*`)` → (*x/y*,*x%y*)
`oct(`*integer*`)` → `str`: octal representation of integer number
`hex(`*integer*`)` → `str`: hexadecimal representation of integer number
Representation formating functions in strings Formating (p6) and Localization (p7).

Math Functions

Standard floating point functions/data in standard `math` module.
`acos(`*x*`)` → `float`: radians angle for *x* cosinus value : [-1…1] →[0…π]
`asin(`*x*`)` → `float`: radians angle for *x* sinus value : [-1…1] →[-π/2…+π/2]
`atan(`*x*`)` → `float`: radians angle for *x* tangent value : [-∞…∞] →]-π/2…+π/2[
`atan2(`*x*`,`*y*`)` → `float`: randians angle for *x/y* tangent value
`ceil(`*x*`)` → `float`: smallest integral value >= *x*
`cos(`*x*`)` → `float`: cosinus value for radians angle *x*
`cosh(`*x*`)` → `float`: hyperbolic cosinus value for radians angle *x*
`exp(`*x*`)` → `float`: exponential of $x = e^x$
`fabs(`*x*`)` → `float`: absolute value of x
`floor(`*x*`)` → `float`: largest integral value <= *x*
`fmod(`*x*`,`*y*`)` → `float`: modulo = remainder of *x/y*
`frexp(`*x*`)` → (`float`,`int`): (m,y) m mantissa of *x*, y exponent of *x* — where $x = m*2^y$
`ldepx(`*x*`,`*i*`)` → `float`: *x* multiplied by 2 raised to *i* power: $x * 2^i$
`log(`*x*`)` → `float`: neperian logarithm of *x*

log10(x) → float: decimal logarithm of x
modf(x) → (float{2}): (f,i) f signed fractional part of x, i signed integer part of x
pow(x,y) → float: x raised to y power (x^y)
sin(x) → float: sinus value for radians angle x
sinh(x) → float: hyperbolic sinus value for radians angle x
sqrt(x) → float: square root of x (√x)
tan(x) → float: tangent value for radians angle x
tanh(x) → float: hyperbolic tangent value for radians angle x
pi → float: value of π (pi=3.1415926535897931)
e → float: value of neperian logarithms base (e=2.7182818284590451)

| Module cmath provides similar functions for complex numbers.

Random Numbers

Randomization functions in standard random module. Module functions use an hidden, shared state, Random type generator (uniform distribution).
Functions also available as methods of Random objects.
seed([x]) ➤ initialize random number generator
random()→ float: random value in [0.0, 1.0[
randint(a,b)→ int: random value in [a, b]
uniform(a, b)→ float: random value in [a, b[
getrandbits(k)→ long: with k random bits
randrange([start,]stop[,step])→ int: random value in range(start, stop, step)
choice(seq)→ value: random item from seq sequence
shuffle(x[,rndfct]) ➤ items of x randomly reordered using rndfct()
sample(population,k)→ list: k random items from polulation
Alternate random distributions : betavariate(alpha,beta), expovariate(lambd), gammavariate(alpha,beta), gauss(mu,sigma), lognormvariate(mu, sigma), normalvariate(mu,sigma), vonmisesvariate(mu,kappa), paretovariate(alpha), weibullvariate(alpha,beta).
Alternate random generator WichmannHill class.
Direct generator manipulation : getstate(), setstate(state), jumpahead(n).
In module os, see :
os.urandom(n) → str: n random bytes suitable for cryptographic use

Other Math Modules

Advanced matrix, algorithms and number crunching in third party modules like numpy (evolution of numarray / Numeric), gmpy (multiprecision arithmetic), DecInt, scipy, pyarray, …

See sites SciPy, BioPython, PyScience,…

Numbers Casts Overriding

__int__(self) → int: called for int(self)
__long__(self) → long: called for long(self)
__float__(self) → float: called for float(self)
__complex__(self) → complex: called for complex(self)
__oct__(self) → str: called for oct(self)
__hex__(self) → str: called for hex(self)
__coerce__(self,other) → value: called for coerce(self,other)

Bit Level Operations

Work with int and long data.

Operators

~x → inverted bits of x
x^y → bitwise exclusive or on x and y
x&y → bitwise and on x and y
x|y → bitwise or on x and y
x<<n → x shifted left by n bits (zeroes inserted)
x>>n → x shifted right by n bits (zeroes inserted)

Binary structures manipulations in standard module struct.

Advanced binary structures mapping and manipulation in third party modules : ctypes, xstruct, pyconstruct, …

Bit Level Overriding

__and__(self,other) → value: for self & other
__or__(self,other) → value: for self | other
__xor__(self,other) → value: for self ^ other
__lshift__(self,other) → value: for self << other
__rshift__(self,other) → value: for self >> other
__invert__(self) → value: for ~self
__iand__(self,other) ➤ called for self &= other
__ior__(self,other) ➤ called for self |= other
__ixor__(self,other) ➤ called for self ^= other
__ilshift__(self,other) ➤ called for self <<= other
__irshift__(self,other) ➤ called for self >>= other

Strings

Simple quoted 'Hello' or double-quoted "Hello".
Use triple [simple|double] quotes for multi-lines strings :

```
"""Hello,
how are you ?"""
```

Strings are immutable (once created a string cannot be modified in place).
Strings can contain binary data, including null chars (chars of code 0).
Strings are sequences, see Indexing (p8) for chars indexation (slicing) and other operations.
chr(code)→ str: string of one char
ord(char)→ int: code
str(expr)→ str: readable textual representation of expr - if available
\`expr\` → str: readable textual representation of expr - if available
repr(expr)→ str: evaluable textual representation of expr - if available

Escape sequences

\a - bell	\v - vertical tab
\b - backspace	\' - single quote
\e - escape	\" - double quote
\f - form feed	\\ - backslash
\n - new line	\ooo - char by octal ooo value
\r - carriage return	\xhh - char by hexadecimal hh value
\t - horizontal tab	\<newline> - continue string on next line.

And for Unicode strings :
\uxxxx - unicode char by 16 bits hexadecimal xxxx value.
\Uxxxxxxxx - unicode char by 32 bits hexadecimal xxxxxxxx value.
\N{name} - unicode char by name in the Unicode database.

| Keep \ escape chars by prefixing string literals with a r (or R) - for 'raw' strings (note : cannot terminate a raw string with a \).

Unicode strings

Quoted as for str, but with a u (or U) prefix before the string :

```
u"Voiçi"
  U"""Une bonne journée
  en perspective."""
```

| Can mix strings prefixs r (or R) and u (or U).

You must define your source file encoding so that Python knows how to convert your source literal strings into internal unicode strings.
unichr(code) → unicode: string of one char
ord(unicode char) → int: unicode code
unicode(object[,encoding[,errors]]) → unicode: unicode
sys.maxunicode → int: maximum unicode code=fct(compile time option)

Unicode Chars Informations

Module unicodedata contains informations about Unicode chars properties, names.
lookup(name) → unicode: unicode char from its name
name(unichr[,default]) → str: unicode name - may raise ValueError
decimal(unichr[,default]) → int: decimal value - may raise ValueError
digit(unichr[,default]) → int: digit value - may raise ValueError
numeric(unichr[,default]) → float: numeric value - may raise ValueError
category(unichr) → str: general unicode category of char
bidirectional(unichr) → str: bidir category of char, may be empty string
combining(unichr) → str/0: canonical combining class of char as integer
east_asian_width(unichr) → str: east asian width
mirrored(unichr) → int: mirrored property in bidi text, 1 if mirrored else 0
decomposition(unichr) → str: decomposition mapping, may be empty str
normalize(form, unistr) → str: normal form of string - form in 'NFC', 'NFKC', 'NFD', 'NFKD'
unidata_version → str: version of Unicode database used

Methods and Functions

From builtins (see also oct and hex functions for integers to strings) :
len(s) → int: number of chars in the string

| Most string methods are also available as functions in the standard string module.

s.capitalize() → string with first char capitalized[1]
s.center(width[,fillchar]) → string centered
s.count(sub[,start[,end]]) → int: count sub occurences
s.decode([encoding[,errors]]) → unicode: text decoded - see encodings (p13)
s.encode([encoding[,errors]]) → str: text encoded - see encodings (p13)
s.endswith(suffix[,start[,end]]) → bool: test text ending
s.expandtabs([tabsize]) → string with tabs replaced by spaces
s.find(sub[,start[,end]]) → int/-1: offset of sub
s.index(sub[,start[,end]]) → int: offset of sub - may raise ValueError
s.isalnum() → bool: non empty string with all alphanumeric chars[1]
s.isalpha() → bool: non empty string with all alphabetic chars[1]
s.isdigit() → bool: non empty string with all digit chars[1]
s.islower() → bool: non empty string with all lower chars[1]
s.isspace() → bool: non empty string with all space chars[1]
s.istitle() → bool: non empty string with titlecase words[1]
s.isupper() → bool: non empty string with all upper chars[1]
s.join(seq) → string: seq[0]+s+seq[1]+s+…+seq[n-1]
s.ljust(width[,fillchar]) → text string left aligned[2]
s.lower() → text string lowered[1]
s.lstrip([chars]) → string text with leading chars[2] removed
s.replace(old,new[,count]) → string with count firsts old replaced by new
s.rfind(sub[,start[,end]]) → int/-1: last offset of sub
s.rindex(sub[,start[end]])→ int: last offset of sub - may raise ValueError
s.rjust(width[,fillchar]) → string text right aligned[2]
s.rsplit([sep[,maxsplit]])→ [string]: rightmost words delim. by sep[2]
s.rstrip([chars]) → string with trailing chars[2] removed
s.split([sep[,maxsplit]]) → [string]: words delimited by sep[2]
s.splitlines([keepends]) → [string]: list of text lines
s.startswith(suffix[,start[,end]]) → bool: test text begining
s.strip([chars]) → string text with leading+trailing chars[2] removed
s.swapcase() → string with case switched[1]
s.title() → string with words capitalized[1]
s.translate(table[,deletechars]) → string: cleaned, converted[3]
s.upper() → string uppered[1]
s.zfill(witdh) → string: string prefixed with zeroes to match width

| [1] Locale dependant for 8 bits strings.
| [2] Default chars/fillchar is space.
| [3] For str table must be a string of 256 chars - see string.maketrans(). For Unicode no deletechars, and table must

be a map of unicode ordinals to unicode ordinals.

Formatting

Use % operator between format string and arguments : *string*%*args*
Formatting *string* contains %[(*name*)][*flag*][*width*][.*precision*]*code*
If not use %(*name*)… → *args* = single value or tuple of values.
If use %(*name*)… → *args* = mapping with *name* as keys.
For mapping, *args* can be an object with __getitem__ method - see Overriding Mapping Operations (p8).

Format char codes

d	signed int. decimal : -324	i	signed int. decimal : -324
o	unsigned octal : 774	u	unsigned decimal 6953
x	unsigned hexa : f3a	X	unsigned hexa : F3A
e	float. point exp. : -3.256e-12	E	float. point exp. : -3.256E-12
f	float. point dec. : -0.0000032	F	float. point dec. : -0.0000032
g	like e or f	G	like E or F
c	character (1 char str or code)	%	%% → %
r	object format like repr(*object*)	s	object format like str(*object*)

Templates

With string.Template objects. Use common $ syntax : $$ ➤ single $; $*name* or ${*name*} ➤ value for *name*.
tmpl = string.Template(*template_string*)
tmpl.substitute(*mapping*[,***kwargs*]) → string: template filled
tmpl.safe_substitute(*mapping*[,***kwargs*]) → string: template filled
tmpl.template → string
Can subclass Template to build your own templating (see doc, sources).
See also modules formatter.

Wrapping

Module textwrap has a TextWrapper class and tool functions.
tw = textwrap.TextWrapper([…]) → new text wrapper using named params as corresponding attributes values
tw.width → int: max length of wrapped lines (default 70)
tw.expand_tabs → bool: replace tabs by *text*.expandtabs() (default True)
tw.replace_whitespace → bool: replace each whitespace by space (default True)
tw.initial_indent → string: prepend to first wrapped line (default '')
tw.subsequent_indent → string: prepend to other wrapped lines (default '')
tw.fix_sentence_endings → bool: try to separate sentences by two spaces (default False)
tw.break_long_words → bool: break words longer than width (default True)
tw.initial_indent → string: prepend to first wrapped line (default '')
tw.wrap(*text*) → [string]: list of text lines, each with max width length - no final newline
tw.fill(*text*) → string: whole text, lines wrapped using newlines
Two convenient functions use temporary TextWrapper, built using named parameters corresponding to attributes.
wrap(*text*[,*width*=70[,…]]) → [string]
fill(*text*[,*width*=70[,…]]) → string
dedent(*text*) → string: remove uniform whitespaces at beginning of *text* lines

Constants

Standard module string provide several constants (do not modify, they are used in string manipulation functions) and some str functions are not available as methods.
ascii_letters → str: lowercase and uppercase chars
ascii_lowercase → str: lowercase a-z chars
ascii_uppercase → str: uppercase A-Z chars
digits → str: 0-9 decimal digit chars
hexdigits → str: 0-9a-fA-F hexadecimal digit chars
letters → str: lowercase and uppercase chars[1]

lowercase → str: lowercase a-z chars[1]
octdigits → str: 0-7 octal digit chars
punctuation → str: ascii chars considered as punctuation in C locale
printable → str: printable chars
uppercase → str: uppercase A-Z chars[1]
whitespace → str: whitespace chars (spc, tab, cr, lf, ff, vt)
capwords(*s*) → str: split → capitalize → join
maketrans(*from*,*to*) → translation table usable in str.translate - *from* and *to* must have same length
[1] Definition is locale dependant.

Regular Expressions

Standard module re has a powerfull regexp engine. See regexp HOWTO at http://www.amk.ca/python/howto/regex/.
Use raw string r"…" notation.
See also external projects pyparsing, PLY (Python Lex-Yacc), tpg (Toy Parser Generator)…

Expressions

Metacharacters : . ^ $ * + ? { } [ ] \ | ( ), may use \ escape.
. ➤ match any character except a newline (including newline with DOTALL option)
^ ➤ match start of string (and start of lines with MULTILINE option)
$ ➤ match end of string (and end of lines with MULTILINE option)
*expr** ➤ match 0 or more repetitions of *expr* (as much as possible)
expr+ ➤ match 1 or more repetitions of *expr* (as much as possible)
expr? ➤ match 0 or 1 *expr*
*expr**? ➤ match like *expr** but as few as possible
expr+? ➤ match like *expr*+ but as few as possible
expr?? ➤ match like *expr*? but as few as possible
expr{*m*} ➤ match *m* repetitions of *expr*
expr{[*m*],[*n*]} ➤ match from *m* to *n* repetitions of *expr*, missing *m* default to 0 and missing *n* default to infinite
expr{[*m*],[*n*]}? ➤ match like *expr*{[*m*],[*n*]} but as few as possible
[*set*] ➤ match one char in the set defined by :
  ^ → at begining, invert set definition
  x-*y* → chars from *x* to *y*
  x → see Escape sequences for strings (p5)
  \\- , \\] → chars - and] (- and] at the beginning match - and] chars)
  x → char *x* (including other re metacharacters)
exprA|*exprB* ➤ match *exprA* or *exprB*, short path evaluation
(*expr*) ➤ match *expr* and build a numbered group
(?[i][L][m][s][u][x]) ➤ (at least one ot iLmsux char) group match empty string, modify options flags for entire expression - see I L M S U X options
(?:*expr*) ➤ match *expr* but dont build a group
(?P<*name*>*expr*) ➤ match *expr* and build a group numbered and named (*name* must be valid Python identifier)
(?P=*name*) ➤ match text matched by earlier group named *name*
(?#*text*) ➤ no match, text is just a comment
(?=*expr*) ➤ match if match *expr* but don't consume input
(?!*expr*) ➤ match if doesn't match *expr* but don't consume input
(?<=*expr*) ➤ match if current position is immediatly preceded by a match for fixed length pattern *expr*
(?<!*expr*) ➤ match if current position is immediatly not preceded by a match for fixed length pattern *expr*
(?(*num/name*)*yesexpr*[|*noexpr*]) ➤ try to match *yesexpr* if group *num/name* exists, else try to match *noexpr*

Escape Sequences

n *nn* ➤ match[3] group number *n* (*nn*) where first $n \neq 0$
ooo \\0*o* ➤ match[3] char with octal value *ooo* (0*o*)
\A ➤ match only at the start of the string
\b ➤ match3 empty string at beginning or end of a word[1+2]
\B ➤ match empty string not at beginning or end of a word[1+2]

\d ➤ match char class decimal digit [0-9]
\D ➤ match char class non-digit [^0-9]
\s ➤ match char class whitespace [\t\n\r\f\v]
\S ➤ match char class non-whitespace [^ \t\n\r\f\v]
\w ➤ match char class alphanumeric [a-zA-Z0-9_]
\W ➤ match char class non-alphanumeric [^a-zA-Z0-9_]
\Z ➤ match end of string
\a \b \f \n \r \t \v \x \\\\ ➤ same as string escapes
c ➤ for other *c* chars, match char *c*

[1] Depends on UNICODE flag.
[2] Depends on LOCALE flag.
[3] When out of char class definition ([…])

Flag Options

IGNORECASE (I) : case insensitive expression - not locale dependant.
LOCALE (L) : make \w \W \b \B locale dependant.
MULTILINE (M) : ^ and $ match begining/end of string and lines. Else ^ and $ match only beginning and end of string.
DOTALL (S) : make . match any char including newline. Else newline excluded.
UNICODE (U) : make \w \W \b \B unicode dependant.
VERBOSE (X) : ignore whitespaces and make # starting comments (except when space and # are escaped or in char class).

Matching and Searching

Can use re functions, or compile expressions into SRE_Pattern objects and use their methods.
See Flag Options supra for *flags* parameters.
search(*pattern*,*string*[,*flags*]) → MatchObject/None: scan throught string to find substrings matching *pattern*
match(*pattern*,*string*[,*flags*]) → MatchObject/None: try to match *string* with *pattern*
split(*pattern*,*string*[,*maxsplit=0*]) → [string]: split *string* by occurences of *pattern* - if *maxsplit* specified, remainder is put in last item of list
findall(*pattern*,*string*[,*flags*]) → [string]/[(string)]: find non-overlapping substrings matching *pattern* - eventually empty matchs - return list of tuples if *pattern* has groups
finditer(*pattern*,*string*[,*flags*]) → iterator over [MatchObject] - same as findall but with an iterator
sub(*pattern*,*repl*,*string*[,*count=0*]) → string: replace substrings matching *pattern* by *repl* - *repl* as string can contain back references[1] to identified substring - repl as fct(MatchObject) return replacement string - pattern may be RE_Pattern object
subn(*pattern*,*repl*,*string*[,*count=0*]) → (string,int): same as sub, 2nd item is count of substitutions
escape(*string*) → string: non-alphanumerics backslashed

If you need to reuse a pattern, compile it one time for all.
pat = re.compile(*pattern*[,*flags*]) → RE_Pattern object
pat.match(*string*[,*pos*[,*endpos*]]) → same as match function[2]
pat.search(*string*[,*pos*[,*endpos*]]) → same as search function[2]
pat.split(*string*[,*maxsplit=0*]) → same as split function[2]
pat.findall(*string*[,*pos*[,*endpos*]]) → same as findall function[2]
pat.finditer(*string*[,*pos*[,*endpos*]]) → same as finditer function[2]
pat.sub(*repl*,*string*[,*count=0*]) → same as sub function
pat.subn(*pattern*,*repl*,*string*[,*count=0*]) → same as subn function
pat.flags → int: flags used at compile time
pat.pattern → string: pattern used at compile time
pat.groupindex → dict: mapping of group names to group numbers

Several functions/methods return MatchObject objects.
m.expand(*template*) → string: do backslash substitution on *template* (like sub method) using match object groups values
m.group([*group*[,…]]) → string/(string): subgroups of the match from numbers or names
m.groups([*default=None*]) → (string): all subgroups of the match - *default* give access to subgroups not in the match
m.groupdict([*default=None*]) → dict: name→subgroup: all named

subgroups of the match - *default* give access to subgroups not in the match
m.**start**([*group=0*]) → int: index of start of substring matched by *group*, -1 if *group* exists but not in match
m.**end**([*group=0*]) → int: index of end of substring matched by *group*, -1 if *group* exists but not in match
m.**span**([*group=0*]) → (int{2}): values of start and end methods for the *group*
m.**pos** → int: pos value of search/match method
m.**endpos** → int: endpos value of search/match method
m.**lastindex** → int/None: index of last matched capturing group
m.**lastgroup** → string/None: name of last matched capturng group
m.**re** → RE_Pattern: pattern used to produce match object
m.**string** → string: string used in match/search to produce match object

[1] Back references extended to \g<*groupnum*> and \g<*groupname*>.
[1] Using part of string between *pos* and *endpos*.
Group number 0 correspond to entire matching.

Localization

Standard module **locale** provide posix locale service (internationalization).
setlocale(*category*[,*locale*]) → current/new settings: if *locale* specified (as string or as tuple(language code, encoding)) then modify locale settings for category and return new one - if *locale* not specified or None, return current locale - not thread safe
localeconv() → dict: database of local conventions
nl_langinfo(*option*) → string: locale-specific informations - not available on all systems - *options* may vary on systems - see options p7
getdefaultlocale([*envvars*]) → (language code, encoding): try to determine default locale settings
getlocale([*category*]) → current LC_* setting for category - category default to LC_CTYPE - for language code and ancoding it may be None
getpreferredencoding([*do_setlocale*]) → str: user preffered encoding for text data - set *do_setlocale* to False to avoid possible call to **setlocale**()
normalize(*localename*) → normalized locale code for *localename* - usable with **setlocale**() - return *localename* if normalization fails
resetlocale([*category*]) ➤ reset locale for category to default setting - category default to LC_ALL
strcoll(*s1*,*s2*) → int: compare two strings - follow LC_COLLATE setting - return 0 if s1==s2, <0 if s1<s2, >0 if s1>s2
strxfrm(*string*) → string:transform string for locale-aware comparison
format(*format*,*val*[,*grouping*]) → string:convert val float using format (% operator conventions) - follow LC_NUMERIC settings (decimal point, + grouping if it is true)
str(float) → string: convert float - follow LC_NUMERIC settings (decimal point)
atof(*string*) → float: convert string to float - follow LC_NUMERIC settings
atoi(*string*) → int: convert string to integer - follow LC_NUMERIC settings
CHAR_MAX → symbolic constant used by **localeconv**()

Categories

LC_CTYPE → character type - case change behaviour
LC_COLLATE → strings sorting - **strcoll**() and **strxfrm**() functions
LC_TIME → time formating - **time.strftime**()
LC_MONETARY → monetary values formating - options from **localeconv**()
LC_MESSAGES → messages display - **os.strerror**() - not for Python messages
LC_NUMERIC → numbers formatting - **format**(), **atoi**(), **atof**() and **str**() of this module (dont modify normal Python number formatting)
LC_ALL → all locales - used to change/retrieve the locale for all categories

nl_langinfo options

key	nl_langinfo() value usage
CODESET	name of character encoding
D_T_FMT	usable as format for **strftime**() for time and date
D_FMT	usable as format for **strftime**() for date
T_FMT	usable as format for **strftime**() for time
T_FMT_AMPM	usable as format for **strftime**() for time in am/pm format
DAY_1…DAY_7	name of the nth day of the week - first day is sunday
ABDAY_1… ABDAY_7	abbreviated name of the nth day of the week - first day is sunday
MON_1… MON_12	name of the nth month
ABMON_1… ABMON_12	abbreviated name of the nth month
RADIXCHAR	radix character (decimal dot/comma/…)
THOUSEP	separator character for thousands
YESEXPR	regular expression (of C library!) usable for yes reply
NOEXPR	regular expression (of C library!) usable for no reply
CRNCYSTR	currency symbol, preceded by - if should appear before the value, by + if should appear after the value, by . if should replace radix character
ERA	era - generally not defined - same as E format in **strftime**()
ERA_YEAR	year in era
ERA_D_T_FMT	usable as format for **strftime**() for date and time with era
ERA_D_FMT	usable as format for **strftime**() for date with era
ALT_DIGITS	up to 100 values representing 0 to 99

localeconv keys

key	meaning
currency_symbol	Local currency symbol for **monetary** values.
decimal_point	Decimal point character for **numbers**.
frac_digits	Number of fractional digits used in local formatting of **monetary** values.
grouping	[int]: relative positions of 'thousands_sep' in **numbers**. CHAR_MAX at the end stop grouping. 0 at the end repeat last group.
int_curr_symbol	International currency symbol of **monetary** values.
int_frac_digits	Number of fractional digits used in international formatting of **monetary** values.
mon_decimal_point	Decimal point used for **monetary** values.
mon_grouping	Equivalent to 'grouping', used for **monetary** values.
mon_thousands_sep	Group separator used for **monetary** values.
n_cs_precedes	True if currency symbol preceed negative **monetary** values, false if it follow.
n_sep_by_space	True if there is a space between currency symbol and negative **monetary** value.
n_sign_posn	Position of negative sign for **monetary** values[1].
negative_sign	Symbol used to annotate a negative **monetary** value.
p_cs_precedes	True if currency symbol preceed positive **monetary** values, false if it follow.
p_sep_by_space	True if there is a space between currency symbol and positive **monetary** value.
p_sign_posn	Position of positive sign for **monetary** values[1].
positive_sign	Symbol used to annotate a positive **monetary** value.
thousands_sep	Character used between groups of digits in **numbers**.

[1] Possible values : 0=currency and value surrounded by parentheses, 1=sign should precede value and currency symbol, 2=sign should follow value and currency symbol, 3=sign should immediately precede value, 4=sign should immediately follow value, LC_MAX=nothing specified in this locale.

Multilingual Support

Standard module **gettext** for internationalization (I18N) and localization (L10N) services - based on GNU gettext API + higher interface. See docs for explanations about tools usage.

Base API

bindtextdomain(*domain*[,*localedir*]) → str: bounded directory - bind domain to localedir directory if specified (used when searching for .mo files)
bind_textdomain_codeset(*domain*[,*codeset*]) → codeset binding: bind domain to codeset if specified - change xxgettext() returned strings encoding
textdomain([*domain*]) → global domain: set global domain if specified and not None
gettext(*message*) → string: localized translation of message - based on current global domain, language, and locale directory - usually aliased as _ in local namespace
lgettext(*message*) → string: like **gettext**(), using preferred encoding
dgettext(*domain*,*message*) → string: like **gettext**(), looking in specified domain.
ldgettext(*domain*,*message*) → string: like **dgettext**(), using preferred encoding
ngettext(*singular*,*plural*,*n*) → string: like **gettext**(), but consider plural forms (see Python and GNU gettext docs)
lngettext(*singular*,*plural*,*n*) → string: like **ngettext**(), using preferred encoding
dngettext(*domain*,*singular*,*plural*,*n*) → string: like **ngettext**(), looking in specified domain.
ldngettext(*domain*,*singular*,*plural*,*n*) → string: like **dngettext**(), using preferred encoding

Generally _ is bound to **gettext.gettext**, and translatable strings are written in sources using _('thestring'). See docs for usage examples.

Class based API

The recommended way. Module **gettext** defines a class Translations, dealing with .mo translation files and supporting str/unicode strings.
find(*domain*[,*localedir*[,*languages*[,*all*]]]) → str/None: .mo file name for translations (search in localedir/language/LC_MESSAGES/domain.mo)
translation(*domain*[,*localedir*[,*languages*[,*class_*[,*fallback*[,*codeset*]]]]]) →Translations: object from class class_ (default to GNUTranslations, constructor take file object as parameter) - if true fallback allow to return a NullTranslations if no .mo file is found, default to false (raise IOError) - codeset change charset used to encode translated strings
install(*domain*[,*localedir*[,*unicode*[,*codeset*]]]) ➤ install _ function in Python's builtin namespace, to use _('thestring')

Null Translations

The NullTranslations is a base class for all Translations.
t.**__init__**([*fp*]) ➤ initialize translations: *fp* is a file object - call **_parse**(*fp*) if it is not None
t.**_parse**(*fp*) ➤ nothing: subclasses override to read data from the file
t.**add_fallback**(*fallback*) ➤ add fallback used if cannot found translation for a message
Define methods **gettext**, **lgettext**, **ngettext**, **lngettext** as in the base API. And define speciale methods **ugettext** and **ungettext** returning unicode strings (other forms return encoded str strings).

Return translated message, forwarding to fallback if it is defined. Overriden in subclasses.
`t.info()`→ return protected `_info` attribute
`t.charset()`→ return protected `_charset` attribute
`t.output_charset()`→ return protected `_output_charset` attribute (defining encoding used to return translated messages)
`t.set_output_charset(`*charset*`)` ➤ set `_output_charset` attribute
`t.install([`*unicode*`])` ➤ bind `_` in builtin namespace to `self.gettext()` or `self.ugettext()` upon unicode (default to false)

GNU Translations

The `GNUTranslations` class (subclass of `NullTranslations`) is based on GNU gettext and .mo files.
Messages ids and texts are coerced to unicode.
Protected `_info` attribute contains message translations.
Translation for empty string return meta-data (see doc).
Define methods `gettext`, `lgettext`, `ugettext`, `ngettext`, `lngettext`, `ungettext` as in `NullTranslations` interface - same rules for return values (`str`/`unicode`). Message translations are searched in catalog, then in fallback if defined, and if no translation is found, message itself is returned (for `n...` methods, return singular forms if *n*=`1` else plural forms).

Containers

Basic containers kind :
-**sequences** (ordered collections) : `list`, `tuple`,`str`, any iterable,…
-**mappings** (unordered key/value) : `dict`…
-**sets** (unordered collections) : `set`, `frozenset`…

Operations on Containers

For strings, items are chars. For mappings, items are keys.
item `in` *container* → `bool`: test *item* ∈ *container*[1]
item `not in` *container* → `bool`: test *item* ∉ *container*[1]
`for` *var* `in` *container*`:` … ➤ iterate *var* over items of *container*
`len(`*container*`)` → int: count number of items in *container*[2]
`max(`*container*`)` → value: biggest item in *container*
`min(`*container*`)` → value: smallest item in *container*
`sum(`*container*`)` → value: sum of items (items must be number-compatible)

[1] For strings test if expr is a substring of sequence.
[2] Container must provide direct length method - no generator.

Copying Containers

Default containers constructors build new container with references to existing objects (shallow copy). To duplicate content too, use standard module `copy`. See Copying Objects (p3).

Overriding Containers Operations

`__len__(self)`→ `int`: called for `len(self)`
`__contains__(self,item)`→ `bool`: called for `item [not] in self`
You can override iterable protocol on containers too.

Sequences

Sequences are ordered collections : `str`, `unicode`, `list`, `tuple`, `buffer`, `xrange`, `array.array`… any user class defining sequences interface, or any iterable data.

Lists & Tuples

Builtin types `list` and `tuple` store sequences of any objects.
Lists are mutable, tuples are immutable.
Declare a list : `[`*item*`[,`…`]]`
Declare a tuple : `(`*item*`[,`…`])`
Notes : `[]` ➤ empty list ;`()` ➤ empty tuple ; `(`*item*`,)` ➤ one item tuple.
`list(`*object*`)` → `list`: new list (cast from object / duplicate existing)
`tuple(`*object*`)` → `tuple`: new tuple (cast from object / duplicate existing)
`range([`*start*`,]`*stop*`[,`*step*`])`→ `[int]`: list, arithmetic progression of integers
`xrange`[1]`([`*start*`,]`*stop*`[,`*step*`])` → `xrange`: object generating arithmetic progression of integers

Unless using a sequence as a mapping key, or ensuring it is immutable data, prefer `list` to `tuple`.
[1] Use in place of range to avoid building huge lists just for indexing.

Operations on Sequences

See Operations on Containers (p8) too.
seq1 `+` *seq2* → concatenation of *seq1* and *seq2*
sequence `*` *n* → concatenation of *sequence* duplicated *n* times
n `*` *sequence* → concatenation of *sequence* duplicated *n* times
`reversed(`*sequence*`)`→ iterator throught *sequence* in reverse order
`sorted(`*sequence*`[,`*cmp*`[,`*key*`[,`*reverse*`]]])`→ `list`: new list, sorted items from iterable - see `list.sorted()`
`filter`[1]`(`*fct*`,`*sequence*`)`→ `list`: new list where fct(item) is `True`. Use `None` fct for a boolean test on items
`map`[1]`(`*fct*`,`*sequence*`,`…`)`→ `list`: new list where i^{th} item is *fct*(i^{th} items of *sequence(s)*)
`reduce(`*fct*`,`*sequence*`[,`*initializer*`])`→ value: fct applied cumulatively to sequence items, f(f(…f(f(f(initializer,a),b),c,…)
`zip`[1]`(`*sequence*`,`…`)`→ `list`: list of tuples, i^{th} tuple contains i^{th} items of each sequences

[1] See Iteration Tools (p9) as replacement (avoid creating a new list).

Indexing

Use index `[`*i*`]` and slice `[`*i*`:`*j*`[:`*step*`]]` syntax. Indexs zero-based. Negative indexs indexing from end. Default step is `1`, can use negative steps.
Sub-sequences indexs between items.

```
l = [e1,e2,e3,…,en-2,en-1,en]
l[0]→ e1          l[0:n]→[e1,e2,e3,…,en-2,en-1,en]
l[1]→ e2          l[:]→[e1,e2,e3,…,en-2,en-1,en]
l[-2]→ en-1       l[i:]→[ei+1,ei+2,ei+3,…,en-1,en]
l[-1]→ en         l[:i]→[e1,e2,…,ei-2,ei-1,ei]
```

items indexs

-n	-n+1	-n+2	…	-2	-1
0	1	2	…	n-2	n-1
e_1	e_2	e_3	**…item…**	e_{n-1}	e_n

0	1	2	3	…	n-2	n-1	n
-n	-n+1	-n+2	-n+3	…	-2	-1	

slicing indexs

Slice objects

Defines index range objects, usable in `[]` notation.
`slice([`*start*`,]`*stop*`[,`*step*`])`→ `slice` object
slice`.indices(`*len*`)`→ `(int{3})`: (start,stop,stride)
Ordered sets of data indexed from 0. Members `start`, `stop`, `step`.

Extended Slicing

Multiple slices notation - corresponding to a selection in a multi-dimension data - can be written using notation like
`[ a , x:y:z , : , : , : , m:n ]`.
Ellipsis notation can be used to fill multiple missing slices, like
`[ a , x:y:z , ... , m:n ]`. See docs.

Three dot notation `...` is replaced internally by `Ellipsis` object.

Operations on mutable sequences

Mutable sequences (ex. `list`) can be modified in place.
Can use mutable sequence indexing in left part of assignment to modify its items : *seq*`[`*index*`]=`*expr* ; *seq*`[`*start*`:`*stop*`]=`*expr* ; *seq*`[`*start*`:`*stop*`:`*step*`]=`*expr*
seq`.append(`*item*`)` ➤ add item at end of sequence
seq`.extend(`*otherseq*`)` ➤ concatenate otherseq at end of sequence
seq`.count(`*expr*`)` → `int`: number of expr items in sequence
seq`.index(`*expr*`[,`*start*`[,`*stop*`]])`→ `int`: first index of expr item
seq`.insert(`*index*`,`*item*`)` ➤ item inserted at index
seq`.remove(`*expr*`)` ➤ remove first expr item from sequence
seq`.pop([`*index*`])` → item: remove and return item at index (default -1)
seq`.reverse()` ➤ items reversed in place
seq`.sort([cmp][,key][,reverse])` ➤ items sorted in place - `cmp` : custom comparison fct(a,b), retval <0 or = 0 or >0 - `key` : name of items attribute to compare - `reverse` : `bool`
`del` *seq*`[`*index*`]` ➤ remove item from sequence
`del` *seq*`[`*start*`:`*stop*`[:`*step*`]]` ➤ remove items from sequence

Overriding Sequences Operations

`__getitem__(self,`*index*[2]`)`→ value: item at *index*, called for `self[`*index*`]`
`__setitem__`[1]`(self,`*index*[2]`,`*value*`)` ➤ set item at *index* to *value*, called for `self[`*index*`]=`*value*
`__delitem__`[1]`(self,`*index*[2]`)` ➤ remove item at *index*, called for `del self[`*index*`]`
[1] Only for mutable sequences.
[2] Parameter index can be a slice [start,stop,step] - replace old __getslice__, __setslice__, __delslice__.
Can also override arithmetic operations `__add__` (concatenation) and `__mul__` (repetition), container operations and object operations.

Mappings (dictionaries)

Builtin type `dict`. Store key:value pairs.
Declare a dictionary : `{` *key*`:`*value* `[,`…`]}` `{}`
`dict()`→ `dict`: empty dictionary (like {})
`dict(**`*kwargs*`)`→ `dict`: from named parameters and their values
`dict(`*iterable*`)`→ `dict`: from (key,value) by iterable
`dict(`*otherdict*`)`→ `dict`: duplicated fro another one (first level)

Operations on Mappings

See Operations on Containers (p8) too, considering operations on keys.
d`[`*key*`]`→ value for *key*[1]
d`[`*key*`]=`*value* ➤ set *d*`[`*key*`]` to *value*
`del` *d*`[`*key*`]` ➤ removes *d*`[`*key*`]` from *d*[1]
d`.fromkeys(`*iterable*`[,`*value*`=None])` → `dict`: with keys from *iterable* and all same value
d`.clear()` ➤ removes all items from *d*
d`.copy()` → `dict`: hallow copy of *d*
d`.has_key(`*k*`)`→ `bool`: test key presence - same as *k* `in` *d*
d`.items()`→ `list`: copy of *d*'s list of (key, item) pairs
d`.keys()`→ `list`: copy of *d*'s list of keys
d`.update(`*otherd*`)` ➤ copy *otherd* pairs into *d*
d`.update(`*iterable*`)` ➤ copy (key,value) pairs into *d*
d`.update(**`*kwargs*`)` ➤ copy name=value pairs into *d*
d`.values()`→ `list`: copy of *d*'s list of values
d`.get(`*key*`,`*defval*`)`→ value: *d*`[`*key*`]` if *key*∈*d*, else *defval*
d`.setdefault(`*key*`[,`*defval*`=None])` → value: if *key*∉*d* set *d*`[`*key*`]=`*defval*, return *d*`[`*key*`]`
d`.iteritems()`→ iterator over (key, value) pairs
d`.iterkeys()`→ iterator over keys
d`.itervalues()`→ iterator over values
d`.pop(`*key*`[,`*defval*`])` → value: del *key* and returns the corresponding value. If *key* is not found, *defval* is returned if given, otherwise `KeyError` is raised
d`.popitem()` → removes and returns an arbitrary (key, value) pair from *d*
[1] If key doesn't exist, raise `KeyError` exception.

Overriding Mapping Operations

`__getitem__(self,`*key*`)`→ value for *key*, called for `self[`*key*`]`
`__setitem__(self,`*key*`,`*value*`)` ➤ set value for *key*, called for `self[`*key*`]=`*value*
`__delitem__(self,`*key*`,`*value*`)` ➤ remove value for key, called for

del self[*key*]

Can also override container operations and object operations.

Other Mappings

For on-disk mappings, see standard module **shelve**, and database modules .

For ordered mappings see third party modules **OrderedDict**.

Sets

Unordered collections of unique items. Frozen sets are immutable once created.

set([*iterable*]) → set: using values from *iterable*

frozenset([*iterable*]) → frozenset: using values from *iterable*

Operations on Sets

See Operations on Containers (p8) too.

s.**issubset**(*others*)→ bool: test *s* ⊂ *others*

s.**issuperset**(*others*)→ bool: test *others* ⊂ *s*

s.**add**(*item*) ➤ add *item* to set

s.**remove**(*item*) ➤ remove *item* from set[1]

s.**clear**() ➤ emoves all items from (not forzen) set

s.**intersection**(*others*)→ set: *s* ∩ *others*

s & *others* → set: *s* ∩ *others*

s.**union**(*others*) → set: *s* ∪ *others*

s | *others* → set: *s* ∪ *others*

s.**difference**(*others*) → set: [x / x∈*s* and x∉*others*]

s - *others* → set: [x / x∈*s* and x∉*others*]

s.**symmetric_difference**(*others*)→ set: [x / x∈*s* xor x∈*others*]

s ^ *others* → set: [x / x∈*s* xor x∈*others*]

s.**copy**()→ set: shallow copy of *s*

s.**update**(*iterable*) ➤ adds all values from *iterable* to *s*

[1] Raise KeyError if object not in set.
Results *set* have same type as *s* object (set/frozenset).

Other Containers Structures, Algorithms

Generally containers follow Python idioms, you can use : **len**(*cont*), *cont*[*i*], for *item* in *cont*:…

Array

Standard module **array** provides efficient array of basic types. It uses compact storage for elements of same type.

Type Codes

n	tc	C type	py type	n	tc	C	py type
1	'b'	signed char	int	1	'B'	unsigned char	int
1	'c'	char	str	2	'u'	unicode char	unicode
2	'h'	signed short	int	2	'H'	unsigned short	int
2	'i'	signed int	int	2	'I'	unsigned int	long
4	'l'	signed long	int	4	'L'	unsigned long	long
4	'f'	float	float	8	'd'	double	float

n=size in bytes, tc=char typecode to use

Functions

array(*tc*,[*iterable*]) → array: with typecode *tc*, initialized from *iterable*

a.**typecode** → str: typecode of *a* data

a.**itemsize** → int: bytes size of *a* data

a.**append**(*expr*) ➤ append item *expr* to end of *a*

a.**extend**(*array*) ➤ append items from another *array*

a.**count**(*expr*) → int: number of *expr* items

a.**index**(*expr*) → int: first index of *expr* item

a.**insert**(*index*,*expr*) ➤ *expr* item inserted at *index*

a.**remove**(*expr*) ➤ remove first *expr* item

a.**pop**([*index*]) → value: return and remove item at *index* (default -1)

a.**reverse**() ➤ items in array are reversed

a.**buffer_info**() → (int{2}): current storage infos (address,items count)

a.**byteswap**() ➤ swap bytes of array items

a.**fromfile**(*f*,*n*) ➤ append n items read from real binary file f[1]

a.**tofile**(*f*) ➤ write all items to real binary file *f*

a.**fromlist**(*list*) ➤ extend array from values in *list*

a.**tolist**() → list: items in a list

a.**fromstring**(*s*) ➤ extend array from values in binary buffer *s* (string)

a.**tostring**() → str: items in binary representation

a.**fromunicode**(*s*) ➤ extend 'u' array from data in unicode stirng

a.**tounicode**() → unicode: convert 'u' array to unicode string

[1] If less items than needed, get available ones then raise EOFError.
Old methods read and write replaced by fromfile and tofile.

Queue

Standard module **collections** provides queues management.

deque([*iterable*])→ deque: initialized from *iterable*

q.**append**(*x*) ➤ add *x* to right side of deque

q.**appendleft**(*x*) ➤ add *x* to left side of deque

q.**clear**() ➤ remove all elements from deque

q.**extend**(*iterable*) ➤ extend right side of deque with *iterable* items

q.**extendleft**(*iterable*) ➤ extend left side of the deque with *iterable* items

q.**pop**() → item: pop and return item from dequeue right side

q.**popleft**() → item: pop and return item from dequeue left side

q.**rotate**(*n*) ➤ rotate deque from n steps, to right if n>0, to left if n<0

Can also use standard operations on sequences : **len**(*q*), **reversed**(*q*), **copy.copy**(*q*), **copy.deepcopy**(*q*), *item* in *q*, *q*[-1], and serialization via pickling protocol.

Priority Queues

Standard module **heapq**. Structure a list as a priority queue.

heapify(*x*) ➤ transform list *x* into heap

heappush(*heap*,*item*) ➤ push *item* onto *heap*

heappop(*heap*)→ item: pop and return smallest item from the *heap*

heapreplace(*heap*,*newitem*)→ item: pop and return smallest item from the *heap*, push *newitem*

nlargest(*n*,*iterable*)→ list: *n* largest from *iterable*

nsmallest(*n*,*iterable*)→ list: *n* smallest items from *iterable*

Sorted List

Standard module **bisect** maintains lists sorted (via basic bisection algo).

bisect_left(*list*,*item*[,*lo*[,*hi*]])→ int: index to insert *item* at leftmost sorted position[1]

bisect_right(*list*,*item*[,*lo*[,*hi*]])→ int: index to insert *item* at rightmost sorted position[1]

bisect(…) ➤ alias for **bisect_right**(…)

insort_left(*list*,*item*[,*lo*[,*hi*]]) ➤ insert *item* at leftmost sorted position[1]

insort_right(*list*,*item*[,*lo*[,*hi*]]) ➤ insert *item* at rightmost sorted position[1]

insort(…) ➤ alias for **insort_right**(…)

[1] With list previously sorted.

Iteration Tools

Standard module **itertools** provides some practical iterators.

chain(*iterable*[,…])→ iterator over items of several iterables

count([*start*])→ iterator over integers from start (default 0)

cycle(*iterable*)→ iterator cycling over *iterable* items

dropwhile(*predicatefct*,*iterable*)→ iterator over items of *iterable* where *predicatefct*(item) is false

groupby(*iterable*[,*keyfct*])→ iterator over (key value,group[1] of items where *keyfct*(item)=key value), default *keyfct* is identity

ifilter(*predicate*,*iterable*)→ iterator over items of *iterable* where *predicatefct*(item) is true - None predicate filter items being true

ifilterfalse(*predicate*,*iterable*)→ iterator over items of *iterable* where *predicatefct*(item) is false - None predicate filter items being false

imap(*function*,*iterable*[,…])→ iterator over *function*(items at same index from *iterables*[2]), None function return tuples items

islice(*iterable*,[*start*,]*stop*[,*step*])→ iterator over items at slice[3] indexs from *iterable*, None stop goes up to end

izip(*iterable*[,…])→ iterator over tuple(items at same index from iterables)

repeat(*object*[,*count*])→ iterator returning object over and over again, up to count times (default to infinite)

starmap(*function*,*iterable*)→ iterator over function(*tuple item from iterable)

takewhile(*predicatefct*,*iterable*)→ iterator over items of iterable where predicatefct(item) is true

tee(*iterable*[,*n*]) → n independent iterators from same iterable[4], default *n*=2

[1] Group of items is internally used - must save it as list if needed after current iteration.
[2] Stop at end of shorter iterable.
[3] Slice parameters cannot be negative.
[4] Don't use iterable out of tee created iterators.

Date & Time

Module time

Standard module **time** defines common functions and data.

Date & Time Data

- *float_time* = float containing seconds from 'epoch' (january 1 1970 on Unix - see **gmtime**(0)), with sub-second precision in decimal part.
- *tuple_time* = tuple containing 9 int (see table).
- *struct_time* = tuple/object with int attributes (see table).

#	attribute	value	#	attribute	value
0	tm_year	int	5	tm_sec	0…61
1	tm_mon	1…12	6	tm_wday	0…6 (monday=0)
2	tm_mday	1…31	7	tm_yday	0…366
3	tm_hour	0…23	8	tm_isdst	0 (no) 1 (yes) -1 (unknown)
4	tm_min	0…59			

- *float_delay* = float containing seconds, with sub-second precision.

DST is local time, UTC is universal (GMT) time.

accept2dyear → [rw] bool: accept two-digit year values (default true), modifiable via environment var PYTHONY2K

altzone → int: offset (pos/neg) in seconds of DST relatively to UTC, in seconds, use only if daylight is true

daylight → int: ≠0 if a DST timezone is defined

timezone → int: offset (pos/neg) in seconds of local (non DST) timezone

tzname → (str{2}): names of local timezone (non-DST, DST)

Functions

asctime([*t*=[2]])→ str: build local time string from t (tuple_time or struct_time)

clock()→ float: processor time in seconds, for accurate relative time measurement

ctime([*secs*=[2]])→ str: build local time string from float_time second

gmtime([*secs*=[2]])→ struct_time: convert float_time to UTC struct_time

localtime([*secs*=[2]])→ struct_time: convert float_time to DST struct_time

mktime(*t*)→ float_time: convert DST t (tuple_time or struct_time) to float_time - may raise OverflowError or ValueError

sleep(*secs*) ➤ execution suspended during *secs* (float_delay) times, maybe less (signal catching), may be more (process/threads scheduling)

strftime(*format*[,*t*=[2]]) → str: build time string from *t* (tuple_time or struct_time) using *format* string (table infra) - may raise ValueError

strptime(*string*[,*format*]) → struct_time: parse *string* using time *format*[1] - may raise ValueError

time() → float_time: current UTC time

tzset() ➤ resets time conversion rules accordingly to environnment variable TZ - unix only, see docs

[1] Default format "%a %b %d %H:%M:%S %Y". Missing values default to (1900, 1, 1, 0, 0, 0, 0, 1, -1)
[2] Param *secs* default to current time, param *t* default to local current time.

Time format strings

%a	Abbreviated weekday name[1].	%A	Full weekday name[1].
%b	Abbreviated month name[1].	%B	Full month name[1].
%c	Appropriate date and time representation[1].	%d	Month day [01,31].
%H	Hour [00,23].	%I	Hour [01,12].
%j	Year day [001,366].	%m	Month [01,12].
%M	Minute [00,59].	%p	AM or PM[1].
%S	Second [00,61].	%U	Year week [00,53] (Sunday based).
%w	Week day [0,6] (0=Sunday).	%W	Year week [00,53] (Monday based).
%x	Appropriate date representation[1].	%X	Appropriate time representation[1].
%y	Year [00,99].	%Y	Year (with century).
%Z	Time zone name (no characters if no time zone exists).	%%	Literal % char.

[1] Locale language representation.

Module datetime

Standard module `datetime` has tools for date/time arithmetics, data extraction and manipulation.
Defines class : `timedelta`, `time`, `date`, `datetime`, [`tzinfo`].

Module timeit

Standard module `timeit` has functions to measure processing time of code. It can be used in scripts (see docs), or directly in command line :

python -mtimeit [-n *N*] [-r *N*] [-s *S*] [-t] [-c] [-h] [*statement* […]]

-n *N* / --number=*N*	execute *statement* *N* times
-r *N* / --repeat=*N*	repeat timer *N* times (default 3)
-s *S* / --setup=*S*	executed *S* once initially (default `pass`)
-t / --time	use `time.time()` (default except Windows)
-c / --clock	use `time.clock()` (default on Windows)
-v / --verbose	print raw timing results - may repeat option
-h / --help	print help and exit

Other Modules

Standard module `calendar` has functions to build calendars.
See also third party module `mxDateTime`.

FILES

Normal file operations use Python `file` objects (or `file`-like objects with same interface). Some functions directly manipulate files path names (strings). Functions mapping low level OS handlers (mainly those in standard `os` module) use numeric file descriptors (*fd* also known as fileno).
Raw data use `str` type (can contain any data byte values, including 0).

File Objects

Standard file type is builtin `file`. It defines the Python file protocol.
Create a file : `file`(*filename*[,*mode*='r'[,*bufsize*]]) → `file` object
Mode flags (combinable) : 'r' read, 'w' write new, 'a' write append, '+' update, 'b' binary[1], 'U' universal newline[2].
Buffer size : 0 unbuffered, 1 line buffered, >1 around that size.

Open() is an alias for file()
[1] Default text mode tries to interpret newline sequences in the file.
[2] Automatically choose newline sequence in CR or LF or CR+LF adapted from file/to platform.

Methods and Functions

f.`close()` ➤ file flushed and no longer usable
f.`fileno()` → `int`: low level file descriptor (fd)
f.`flush()` ➤ buffers written to file on disk
f.`isatty()` → `bool`: indicator file is a terminal
f.`read`([*size*]) → `str`: block of data read from file
f.`readline()` → `str`: next line read from file, end of line removed
f.`readlines()` → [string]: list of all lines read from file, end of lines removed
f.`seek`(*offset*[,*whence*=0]) ➤ modify current position in file - *whence*: 0 from start, 1 from current, 2 from end
f.`tell()` → `int`: current position in file
f.`write`(*string*) ➤ data written to file
f.`writelines`(*listofstrings*) ➤ data written to file (no end of line added)
`for` *line* `in` *f* :… ➤ iterate *line* over lines of *f*

Old method `xreadlines` replaced by iteration on file object.
For optimized direct access to random lines in text files, see module `linecache`.

Attributes

f.`closed` → `bool`: indicator file has been closed
f.`encoding` → `str`/`None`: file content encoding
f.`name` → `str`: name of the file
f.`newlines` → `str`/`tuple` of `str`/`None`: encountered newlines chars
f.`softspace` → `bool`: indicator to use soft space with `print` in file

Low-level Files

Base low-level functions are in standard module `os`.

Careful of clash with builtins with `os.open` name.

`open`(*path*,*flags*[,*mode*=0777]) → `int` (fd): open file *path* - see *flags* infra - *mode* masked out with umask
`fdopen`(*fd*[,*mode*[,*bufsize*]]) → `file`: build a `file` connected to *fd* - *mode* and *bufsize* as for builtin `open()` + *mode* must start with `r` or `w` or `a`
`dup`(*fd*) → `int` (fd): duplicate file descriptor *fd*
`dup2`(*fd*,*fd2*) → `int` (fd): duplicate file descriptor *fd* into *fd2*, previously closing *fd2* if necessary
`close`(*fd*) ➤ close file descriptor
`read`(*fd*,*n*) → `str`: read as most *n* bytes from *fd* file - return empty string if end of file reached
`write`(*fd*,*str*) → `int`: write *str* to *fd* file - return number of bytes actually written
`lseek`(*fd*,*pos*,*how*) ➤ set file descriptor position - *how*: 0 from start, 1 from current, 2 from end
`fdatasync`(*fd*) ➤ flush file data to disk - don't force update metadata (Unix)
`fsync`(*fd*) ➤ force low level OS buffers to be written
`ftruncate`(*fd*,*length*) ➤ truncate file descriptor to at most length (Unix)

Open Flags

Constants defined in `os` module, use bit-wise OR (x|y|z) to mix them.
`O_RDONLY` → read only
`O_WRONLY` → write only
`O_RDWR` → read/write
`O_APPEND` → append each write to end
`O_CREAT` → create new file (remove existing)
`O_EXCL` → with `O_CREAT`, fail if file exist (Unix)
`O_TRUNC` → reset existing file to zero size
`O_DSYNC` → xxxxxx (Unix)
`O_RSYNC` → xxxxxx (Unix)
`O_SYNC` → return from IO when data are physically written (Unix)
`O_NDELAY` → return immediatly (don't block caller during IO) (Unix)
`O_NONBLOCK` → same as `O_NDELAY` (Unix)
`O_NOCTTY` → terminal device file can't become process tty (Unix)
`O_BINARY` → don't process end of lines (cf+lf from/to cr) (Windows)
`O_NOINHERIT` → xxxxxx (Windows)
`O_SHORT_LIVED` → xxxxxx (Windows)
`O_TEMPORARY` → xxxxxx (Windows)
`O_RANDOM` → xxxxxx (Windows)
`O_SEQUENTIAL` → xxxxxx (Windows)
`O_TEXT` → xxxxxx (Windows)

Pipes

For standard process redirection using pipes, see also Simple External Process Control (p14).
`os.pipe()` → ((`int`{2}){2}): create pair (fdmaster,fdslav) of fd (read,write) for a pipe
`os.mkfifo`(*path*[,*mode*=0666]) ➤ create named pipe *path* - *mode* masked out with umask - don't open it (Unix)
Use `os` functions on file descriptors.

In-memory Files

Memory Buffer Files

Use standard modules `StringIO` and `cStringIO` to build file-like objects storing data in memory.
f = `StringIO.StringIO()`

Build a file-like in memory.

f.`write`(*string*) ➤ data written to file
f.…other file writing methods…
f.`getvalue()` → `str`: current data written to file
f.`close()` ➤ file no longer usable, free buffer
cStringIO is a compiled (more efficient) version of StringIO for writing. Optional argument allows to build memory files to read from too.
f = `cStringIO.StringIO`([*string*])
f.`read`([*size*]) → `str`: block of data read from 'file' (string)
f.…other file reading methods…

Memory Mapped Files (OS level)

Standard module `mmap` manage memory-mapped files, usable as file-like objects and as mutable string-like objects.
To build a memory map :
mm = `mmap.mmap`(*fileno*,*length*[,*tagname*[,*access*]]) [windows]
mm = `mmap.mmap`(*fileno*,*length*[,*flags*[,*prot*[,*access*]]]) [unix]
Use an os file descriptor (from `os.open()` or from file-object's `fileno()`) for a file opened for update.
Length specify amount of bytes to map. On windows, file may be extended to that length if it is shorter, it can't be empty, and 0 correspond to maximum length for the file.
Access (keyword param) : `ACCESS_READ` (readonly), `ACCESS_WRITE` (write-through, default on Windows), or `ACCESS_COPY` (copy-on-write).
On Windows, tagname allow to identify different mappings against same file (default to None).
On Unix, flags : `MAP_PRIVATE` (copy-on-write private to process) or `MAP_SHARED` (default). And prot (memory protection mask) : `PROT_READ` or `PROT_WRITE`, default is `PROT_READ|PROT_WRITE`. If use prot+flags params, don't use access param.
mm.`close()` ➤ mmap file no longer usable
mm.`find`(*string*[,*start*=0]) → `int`: offset / -1
mm.`flush`([*offset*,*size*]) ➤ write changes to disk
mm.`move`(*dest*,*src*,*count*) ➤ copy data in file
mm.`read`([*size*]) → `str`: block of data read from mmap file[1]
mm.`read_byte()` → `str`: next one byte from mmap file[1]
mm.`readline()` → `str`: next line read from file, end of line is not removed[1]
mm.`resize`(*newsize*) ➤ writable mmap file resizer
mm.`seek`(*offset*[,*whence*=0]) ➤ modify current position in mmap file - *whence*: 0 from start, 1 from current, 2 from end
mm.`size()` → `int`: length of the real os file
mm.`tell()` → `int`: current position in mmap file
mm.`write`(*string*) ➤ data written to mmapfile[1]
mm.`write_byte`(*byte*) ➤ `str` of one char (byte) data written to mmap file[1]

[1] File-like methods use and move file seek position.

Files Informations

Functions to set/get files informations are in `os` and in `os.path` module, some in `shutil` module. Constants flags are defined in standard `stat` module.

> Some functions accessing process environment data (ex. current working directory) are documented in Process section.

`os.access(`*path*`,`*mode*`)`→ `bool`: test for *path* access with *mode* using real uid/gid - mode in `F_OK`, `R_OK`, `W_OK`, `X_OK`
`os.F_OK` → access mode to test path existence
`os.R_OK` → access mode to test path readable
`os.W_OK` → access mode to test path writable
`os.X_OK` → access mode to test path executable
`os.chmod(`*path*`,`*mode*`)` ➤ change *mode* of *path* - *mode* use `stat.S_*` constants
`os.chown(`*path*`, `*uid*`, `*gid*`)` ➤ change *path* owner and group (Unix)
`os.lchown(`*path*`, `*uid*`, `*gid*`)` ➤ change path owner and group - don't follow symlinks(Unix)
`os.fstat(`*fd*`)`→ `int`: status for file descriptor
`os.fstatvfs(`*fd*`)`→ `statvfs_result`: informations about file system containing file descriptor (Unix)
`os.stat(`*path*`)`→ stat structure object: file system informations (Unix)
`os.lstat(`*path*`)`→ stat structure object: file system informations (Unix) - dont follow symlinks
`os.stat_float_times([`*newvalue*`])`→ `bool`: test/set `stat` function time stamps data type - avoid setting new value
`os.statvfs(`*path*`)`→ `statvfs_result`: informations about file system containing path (Unix)
`os.utime(`*path*`,`*times*`)` ➤ set access and modification times of file path - times=(atime,mtime) (numbers) - times=`None` use current time
`os.fpathconf(`*fd*`,`*name*`)` → `str` / `int`: system configuration information about file referenced by file descriptor - see platform documentation and `pathconf_names` variable - name `str` or `int` (Unix)
`os.pathconf(`*path*`,`*name*`)`→ `str` / `int`: system configuration information about file referenced by file descriptor - see platform documentation and `pathconf_names` variable - name `str` or `int` (Unix)
`os.pathconf_names` → `dict`: name → index - names accepted by `pathconf` and `fpathconf` → corresponding index on host (Unix)
`os.path.exists(`*path*`)`→ `bool`: test existing path - no broken symlinks
`os.path.lexists(`*path*`)`→ `bool`: test existing path - allow broken symlinks
`os.path.getatime(`*path*`)`→ float_time: last access time of path
`os.path.getmtime(`*path*`)`→ float_time: last modification time of path
`os.path.getctime(`*path*`)`→ float_time: creation time (windows) or last modification time (unix) of path
`os.path.getsize(`*path*`)`→ `int`: bytes size of path file
`os.path.isabs(`*path*`)`→ `bool`: test absolute
`os.path.isfile(`*path*`)`→ `bool`: test regular file (follow symlinks)
`os.path.isdir(`*path*`)`→ `bool`: test existing directory (follow symlinks)
`os.path.islink(`*path*`)`→ `bool`: test symlink
`os.path.ismount(`*path*`)`→ `bool`: test mount point
`os.path.samefile(`*path1*`,`*path2*`)`→ `bool`: test refer to same real file (unix,macos)
`os.path.sameopenfile(`*f1*`,`*f2*`)`→ `bool`: test opened files refer to same real file (unix,macos)
`os.path.samestat(`*stat1*`,`*stat2*`)`→ `bool`: test stat tuples refer to same file *(unix,macos)*
`shutil.copymode(`*srcpath*`,`*dstpath*`)` ➤ copy normal file permission bits
`shutil.copystat(`*srcpath*`,`*dstpath*`)` ➤ copy normal file permission bits and last access and modification times

Stat Structures

`stat_result` is returned by `stat` and `lstat` functions, usable as a tuple and as object with attributes :

#	attribute	usage
0	`st_mode`	protection bits
1	`st_ino`	inode number
2	`st_dev`	device
3	`st_nlink`	number of hard links
4	`st_uid`	user ID of owner
5	`st_gid`	group ID of owner
6	`st_size`	size of file, in bytes
7	`st_atime`	time of most recent access
8	`st_mtime`	time of most recent content modification
9	`st_ctime`	time of most recent metadata change on Unix, time of creation on Windows
	`st_blocks`	number of blocks allocated for file (Unix)
	`st_blksize`	filesystem blocksize (Unix)
	`st_rdev`	type of device if an inode device (Unix)
	`st_rsize`	size of resource fork, in bytes(MacOS)
	`st_creator`	file creator code (MacOS)
	`st_type`	file type code (MacOS)

`statvfs_result` is returned by `fstatvfs`and `statvfs` functions, usable as a tuple (use `statvfs` variable indexs) and as an object with attributes :

#	attribute	index var	usage
0	`f_bsize`	`F_BSIZE`	preferred file system block size
1	`f_frsize`	`F_FRSIZE`	fundamental file system block size
2	`f_blocks`	`F_BLOCKS`	total number of blocks in the filesystem
3	`f_bfree`	`F_BFREE`	total number of free blocks
4	`f_bavail`	`F_BAVAIL`	free blocks available to non-super user
5	`f_files`	`F_FILES`	total number of file nodes
6	`f_ffree`	`F_FFREE`	total number of free file nodes
7	`f_favail`	`F_FAVAIL`	free nodes available to non-super user
8	`f_flag`	`F_FLAG`	flags - see host statvfs() man page
9	`f_namemax`	`F_NAMEMAX`	maximum file name length

Stat Constants

Defined in standard `stat` module.

`S_ISUID` → xxxxx
`S_ISGID` → xxxxx
`S_ENFMT` → xxxxx
`S_ISVTX` → xxxxx
`S_IREAD` → 00400 user can read
`S_IWRITE` → 00200 user can write
`S_IEXEC` → 00100 user can execute
`S_IRWXU` → 00700 user can read+write+execute
`S_IRUSR` → 00400 user can read
`S_IWUSR` → 00200 user can write
`S_IXUSR` → 00100 user can execute
`S_IRWXG` → 00070 group can read+write+execute
`S_IRGRP` → 00040 group can read
`S_IWGRP` → 00020 group can write
`S_IXGRP` → 00010 group can execute
`S_IRWXO` → 00007 everybody can read+write+execute
`S_IROTH` → 00004 everybody can read
`S_IWOTH` → 00002 everybody can write
`S_IXOTH` → 00001 everybody can execute

Terminal Operations

`os.openpty()`→ `(int{2})`: open pseudo-terminal[1] pair (fdmaster,fdslave)=(pty,tty) (Unix)
`os.ttyname(`*fd*`)`→ `str`: terminal device associated to fd (Unix)
`os.isatty(`*fd*`)`→ `bool`: test file descriptor is a tty-like (Unix)
`os.tcsetpgrp(`*fd*`,`*pg*`)` ➤ set process group id associted with terminal fd (Unix)
`os.tcgetpgrp(`*fd*`)`→ `int`: process group associated with terminal fd (Unix)

See also standard modules `tty` and `pty`. For user-interface control on text terminal , see standard package `curses` and its sub-modules.

Temporary Files

Use standard `tempfile` module. It defines several functions to make life easier and more secure.
`TemporaryFile([`*mode='w+b'*`[,`*bufsize=-1*`[,`*suffix*`[,`*prefix*`[,`*dir*`]]]]])`
→ file/file-like: temp file - removed on close - not necessary visible in file-system - *dir* and *prefix* as for `mkstemp`
`NamedTemporaryFile([`*mode='w+b'*`[,`*bufsize=-1*`[,`*suffix*`[,`*prefix*`[,`*dir*`]]]]])`
→ file/file-like: like `TemporaryFile` - file visible in file-system
`mkstemp([`*suffix*`[,`*prefix*`[,`*dir*`[,`*text*`]]]])`→ `(int,str)`: (fd,path) of new temporaty file - no race condition - only creator can read/write - no executable bit - not automatically deleted - binary mode unless text specified
`mkdtemp([`*suffix*`[,`*prefix*`[,`*dir*`]]])`→ `str`: path of new temporary directory created - no race condition - only creator can read/write/search - not automatically deleted
`gettempdir()` → `str`: default directory for temporary files
`gettempprefix()` → `str`: default filename prefix for temporary files

> Other functions in `tempfile` and `os` modules are kept for code compatibility, but are considered not enough secured. Also `tempdir` and `template` data in `tempfile` - which should not be used directly.

Path Manipulations

Path manipulation functions are in standard `os.path` module.
`supports_unicode_filenames` → `bool`: unicode usable for file names
`abspath(`*path*`)`→ `str`: normalized absolutized pathname
`basename(`*path*`)`→ `str`: file name part of *path*
`commonprefix(`*pathlist*`)` → `str`: longest common path prefix (char-by-char)
`dirname(`*path*`)`→ `str`: directory name of pathname
`join(`*path*`[,`*...*`])`→ `str`: concatenate path components
`normcase(`*path*`)`→ `str`: normalize path case for platform (see doc)
`normpath(`*path*`)`→ `str`: normalize path (// /./ /../), on windows /→ \
`realpath(`*path*`)`→ `str`: canonical path (remove symlinks) (unix)
`split(`*path*`)`→ `(str{2})`: split into (head, last pathname component)
`splitdrive(`*path*`)`→ `(str{2})`: split into (drive, tail)
`splitext(`*path*`)`→ `(str{2})`: split into (root, ext)

Host Specific Path Data

`sys.getfilesystemencoding()` → `str`: name of encoding used by system for filenames
Following data are in `os` and in `os.path`.
`curdir` → `str`: string used to refer to current directory
`pardir` → `str`: string used to refer to parent directory
`sep` → `str`: char used to separate pathname components
`altsep` → `str`: alternative char used to separate pathname components
`extsep` → `str`: char used to separate base filename from extension
`pathsep` → `str`: conventional char to separate different paths

Directories

`os.listdir(`*path*`)`→ `[str]/[unicode]`: list names in *path* directory - without `.` and `..` - arbitrary order - path string type → item strings type
`os.mkdir(`*path*`[,`*mode=0777*`])` ➤ create directory *path* - *mode* masked out with umask
`os.makedirs(`*path*`[,`*mode=0777*`])` ➤ create directory *path*, recursively - *mode* masked out with umask - don't handle Windows' UNC path
`os.rmdir(`*path*`)` ➤ remove directory path
`os.removedirs(`*path*`)` ➤ remove directories, recursively
`os.walk(`*top*`[,`*topdown*`=True [,`*onerror=None*`]])`→ iterable: go throught dirs under *top*, for each dir yield tuple(dirpath, dirnames, filenames) - *onerror*=fct(os.error) - see docs
`os.path.walk(`*path*`,`*visit*`,`*arg*`)` ➤ call *visit*(*arg*,dirname,names) for dirs rooted at *path* - may modify names (files list) to influence walk, may prefer to use `os.walk`

Special Files

os.link(*src*,*dst*) ➤ create hard link named *dst* referencing *src* (Unix)
os.symlink(*src*,*dst*) ➤ create symbolic link named *dst* pointing to *src* (Unix)
os.readlink(*path*)→ str: *path* pointed to by symbolic link
os.mknod(*path*[,*mode=0666*,*device*]) ➤ create FS node (file, device special file, named pipe) - *mode* = permissions | nodetype - node type in **S_IFREG**, **S_IFREG**, **S_IFCHR**, **S_IFBLK**, and **S_IFIFO** defined in **stat** module
os.major(*device*) → int: raw device major number
os.minor(*device*) → int: raw device minor number
os.makedev(*major*,*minor*) ➤ compose raw device from major and minor numbers

Copying, Moving, Removing

os.remove(*path*) ➤ remove file *path* (not directory)
os.rename(*src*,*dst*) ➤ rename *src* to *dst* - on same filesystem- may remove existing dst file
os.renames(*old*,*new*) ➤ rename *old* to *new*, recursively - try to create intermediate directories
os.unlink(*path*) ➤ remove file *path* (not directory) - same as **remove**

Standard module **shutil** provides high level functions on files and directories.
copyfile(*src*,*dst*) ➤ copy normal file content - overwrite destination[2].
copyfileobj(*fsrc*,*fdst*[,*length=16kb*]) ➤ copy file-like object content by blocks of length size (<0=one chunk)
copy(*src*,*dst*) ➤ copy normal file content to file/directory[2] - in case of directory use same basename as src - overwrite destination - copy permission bits.
copy2(*src*,*dst*) ➤ same as **copy** + copy last access and modification times[2].
copytree(*src*,*dst*[,*symlinks*=False]) ➤ recursively copy directory tree - destination must be new - files copied via **copy** - if *symlinks* is False, copy symbolic links files content, else just make symbolic links.[1]
rmtree(*path*[,*ignore_errors*=False[,*onerror*=None]]) ➤ recursively delete directory tree - *onerror*=fct(fctref, path, excinfo).[1]
move(*src*,*dst*) ➤ recursively move file or directory tree - may rename or copy.[1]

[1] May raise **shutil**.Error exception.
[2] Params src and dst are files path names.

Encoded Files

Standard module **codecs** have functions and objects to transparently process encoded files (used internally as unicode files).
codecs.open(*filename*,*mode*[,*encoding*[,*errors*[,*buffering*]]])→ file-like EncodedFile object with transparent encoding/decoding
codecs.EncodedFile(*file*,*input*[,*output*[,*errors*]]) → file-like wrapper around file, decode from input encoding and encode to output encoding
codecs.BOM → str: alias for **BOM_UTF16**
codecs.BOM_BE → str: alias for **BOM_UTF16_BE**
codecs.BOM_LE → str: alias for **BOM_UTF16_LE**
codecs.BOM_UTF8 → str: '\xef\xbb\xbf'
codecs.BOM_UTF16 → str: alias for **BOM_UTF16_LE** or **BOM_UTF16_BE**
codecs.BOM_UTF16_BE → str: '\xfe\xff'
codecs.BOM_UTF16_LE → str: '\xff\xfe'
codecs.BOM_UTF32 → str: alias for **BOM_UTF32_LE** or **BOM_UTF32_BE**
codecs.BOM_UTF32_BE → str: '\x00\x00\xfe\xff'
codecs.BOM_UTF32_LE → str: '\xff\xfe\x00\x00'

See Encoding - Decoding (p13) for details about *encoding* and *errors*.

Serialization

Standard modules **pickle** and **cPickle** (speed up to 1000x) have support for data serialization of objects hierarchies.
See Python documentation.
See also module **marshal** (read/write of Python data in platform independant binary format - but can broke format between releases).

Persistence

Standard module **shelve** use pickling protocol to store objects in DBM files (see p17) and access them via a dictionnary-like interface with keys as str.
open(*filename*[,*flag*[,*protocol*[,*writeback*[,*binary*]]]])→ dictionary-like object - *flag* as **anydbm.open** (p17), default to 'c' - *protocol* default to 0 (ascii format) - *writeback*: cache accessed entries in memory and written them back at close time, default to False - *binary* is deprecated, use *protocol*.

Configuration Files

Standard module **ConfigParser**. It uses standard .INI files to store configudation data :

```
[section]
name:value
name=value
```

Values can contain %(name)s references which may be expanded using values in same section or in *defaults*
and ; start comment lines.

Module defines 3 configuration classes with different data access level :

```
RawConfigParser
  ConfigParser
    SafeConfigParser
```

rp=**RawConfigParser**([*defaults*]) → RawConfigParser
cp=**ConfigParser**([*defaults*]) → ConfigParser
sp=**SafeConfigParser**([*defaults*]) → SafeConfigParser
In the three constructors, *defaults* is a dict of option:value for references expansion.
MAX_INTERPOLATION_DEPTH → int: max recursive depth for get() when raw parameter is false
DEFAULTSECT → str: name of defaut section

Raw Interface

rp.**defaults**()→ dict: default values for references expansion
rp.**sections**()→ [string]: list sections in config (without DEFAULT)
rp.**add_section**(*section*) ➤ add a new *section* - may raise DuplicateSectionError
rp.**has_section**(*section*)→ bool: test if section exists - cant test for DEFAULT
rp.**options**(*section*)→ [string]: list options in section
rp.**has_option**(*section*,*option*)→ bool: test if section and option exists
rp.**read**([*filename*]/*filename*)→ [filename]: try to load configuration data from files (continue if fail) - return names of loaded files
rp.**readfp**(*fp*[,*filename*]) ➤ load configuration data from file/file-like
rp.**get**(*section*,*option*)→ str: option value
rp.**getint**(*section*,*option*)→ int: coerce option value to int
rp.**getfloat**(*section*,*option*)→ float: coerce option value to float
rp.**getboolean**(*section*,*option*)→ bool: coerce option value to bool - True is strings 1 yes true on - False is strings 0 no false off - may raise ValueError
rp.**items**(*section*)→[(name,value)]: options in the section
rp.**set**(*section*,*option*,*value*) ➤ set *option* to *string* value in *section* - may raise NoSectionError
rp.**write**(*fileobject*) ➤ write configuration data to file
rp.**remove_option**(*section*,*option*)→ bool: return True if there was such option - may raise NoSectionError
rp.**remove_section**(*section*)→ bool: return True if there was such section
rp.**optionxform**(*option*)→ str: normalized internal form of option

Normal Interface

cp.**get**(*section*,*option*[,*raw*[,*vars*]])→ string: value for *option* in *section* - % interpolation expanded unless *raw* is true - *vars* is a dict of additional defaults - reference expansion names are processed by **optionxform**() for matching
cp.**items**(*section*[,*raw*[,*vars*]])→ [(name,value)]: for given section - *raw* and *vars* as in **get**()

Safe Interface

sp.**set**(*section*,*option*,*value*) ➤ set value string for section and option

Exceptions

```
(Exception)
  Error
  ParsingError
    NoSectionError
    DuplicateSectionError
    MissingSectionHeaderError
    NoOptionError
    InterpolationError
      InterpolationDepthError
      InterpolationMissingOptionError
      InterpolationSyntaxError
```

For similar file format supporting nested subsections, see **ConfigObj** config parser. For windows users, standard module **_winreg**.
For text-file configs, can use XML tools, and see also third party YAML parsers like **PyYaml**.

Exceptions

Standard exceptions defined in **exceptions** module, and available in current scope.
All exceptions must be subclasses of Exception root class.
Use standard exceptions if their meaning correspond to you errors.
Subclass standard exceptions when needed.

Standard Exception Classes

Exception
- StopIteration — iterator's next(), no more value.
- SystemExit — sys.exit() called
- StandardError — built-in exceptions
  - ArithmeticError — arithmetic errors.
    - FloatingPointError
    - OverflowError
    - ZeroDivisionError
  - AssertionError — assert *cond*[,*message*] failed.
  - AttributeError — attribute set/get failed.
  - EnvironmentError — host system error - see arg tuple attribute
    - IOError
    - OSError
      - WindowsError — Windows error codes.
  - EOFError — end-of-file with **input**() or **raw_input**().
  - ImportError
  - KeyboardInterrupt — user interrupt (Ctrl-C).
  - LookupError
    - IndexError — non-existent sequence index.
    - KeyError — non-existent mapping key.
  - MemoryError
  - NameError — non-existent name in current scope.
    - UnboundLocalError — reference to an unassigned local variable.
  - ReferenceError — try accessing weak-ref disposed object.
  - RuntimeError — (prefer defining ad-hoc subclasses).
    - NotImplementedError
  - SyntaxError
    - IndentationError
    - TabError
  - SystemError — a bug… in Python.
  - TypeError
  - ValueError — good type, but bad value.
    - UnicodeError
  - Warning — warnings superclass (see Warnings infra)
    - UserWarning

PendingDeprecationWarning
DeprecationWarning
SyntaxWarning
RuntimeWarning

Warnings

Warnings must be subclasses of `Warning` root class.
Standard **warnings** module control processing of warning exceptions.

warn(*message*[,*category*[,*stacklevel*]])
warn_explicit(*message*,*category*,*filename*,*lineno*[,*module*[,*registry*]])
showwarning(*message*,*category*,*filename*,*lineno*[,*file*])
formatwarning(*message*,*category*,*filename*,*lineno*)
filterwarnings(*action*[,*message*[,*category*[,*module*[,*lineno*[,*append*]]]]])
resetwarnings()
sys.warnoptions

Exceptions Processing

sys.exc_info() → (type,value,traceback) for current exception[1]
sys.exc_clear() ➤ current exception related informations cleared
sys.excepthook → (rw) fct(type, value, traceback) called for uncaught exceptions
sys.__excepthook__ → backup of original excepthook function
sys.tracebacklimit → `int`: (rw) maximum levels of traceback printed, <=0 for none

[1] Or (None,None,None) if no running exception.

Standard module **traceback** has tools to process and format these informations.

Encoding - Decoding

Standard module **codecs** provide base support for encoding / decoding data. This is used for character encodings, but also for data compression (zip, bz2) or data representation (uu, hex).
See Unicode strings (p5), Source encodings (p3).
See functions, classes and constants for files encoding in Encoded Files (p12).
Module **encodings.aliases**.

Threads & Synchronization

Python threads use native threads. A global mutex (the GIL) lock interpreter data during Python virtual instructions execution (it is unlocked during I/O or long computation in native code). Check for thread switching and signal processing is performed at regular interval.

sys.getcheckinterval() → `int`: current thread switching check interval[1]
sys.setcheckinterval(*interval*) ➤ set hread switching check interval[1]

[1] Expressed in number of Python virtual instructions.

Threading Functions

Use standard high level module **threading** which provides several classes : `Thread`, `local` (for thread local storage), `Event`, `Lock` and `RLock` (mutex), `Semaphore` and `BoudedSemaphore`, `Timer`.
Module **threading** also provides functions :

activeCount() → `int`: number of currently active threads
currentThread() → `Thread`: current running thread
enumerate() → [`Thread`]: list of active threads
settrace(func) ➤ install trace function called before threads run methods
setprofile(func) ➤ install profile function called before threads run methods

Standard module **thread** supports low level thread management. Use modules **dummy_thread** and **dummy_threading** on platforms without multithreading.

Threads

Class **threading**.`Thread` is used to create new execution path in current process. It must be called with keyword arguments. Specify thread code with a callable *target* param or by overriding **run** method (remember calling inherited **__init__** in subclasses), give arguments in *args* and *kwargs* (tuple and dict), give a *name* to identify the thread - *group* currently not used (None).

th = **threading.Thread**(*group*,*target*,*name*,*args*,*kwargs*)
th.**start**() ➤ start thread activity (in another thread)
th.**run**() ➤ thread code to execute - call target if not overriden
th.**join**([*timeout*]) ➤ wait for th termination or timeout elapsed (float_delay, default to `None` for infinite)
th.**getName**() → `str`: thread associated name
th.**setName**(*name*) ➤ set thread associated name (initial name set by class)
th.**isAlive**() → `bool`: test thread alive (started and run() not terminated)
th.**isDaemon**() → `bool`: test thread have daemon flag
th.**setDaemon**(*daemonic*) ➤ set thread daemon flag - must be called before start. Initial flag inherited from creating thread. Python process exit only after last non-daemon thread termination.

A thread can't be killed or paused externally by another thread.

Thread Local Storage

Class **threading**.`local` attributes values are thread local.
Subclass it or use it as a namespace.

```
tlsdata = threading.local()
tlsdata.x = 1
```

Delayed Start Thread

Class **threading**.`Timer` is a subclass of `Thread` which effectively run after a specified interval from its start.

t = **threading.Timer**(*interval*,*function*,*args*=[],*kwargs*={})
t.**cancel**() ➤ timer will never run - must not be already running

Create a timer that will run function with arguments args and keyword arguments kwargs, after interval seconds have passed.

Mutual Exclusion

Classes **threading**.`Lock` and **threading**.`RLock` provide mutual exclusion between threads. `Lock` doesn't allow a thread to re-acquire a lock it already owns, `RLock` does (reentrant-lock).

lock = **threading.Lock**()
lock = **threading.RLock**()
lock.**acquire**([*blocking*]) → `bool`/`None`: acquire the lock. *blocking* unspecified : wait & return `None` ; *blocking* true : wait & return `True` ; *blocking* false : don't wait (try) & return `True`/`False`
lock.**release**() ➤ unlock a previously acquired lock

Must release a lock same times as it was acquired.
Good practice to **acquire**/**release** locks in `try`/`finally` blocks.
For portable inter-process mutex, see third party `glock.py` module.

Events

Class **threading**.`Event` is a synchronisation flag with thread blocking mechanism to wait for the flag.

evt = **threading.Event**() ➤ new event, with internal flag set to False
evt.**isSet**() → `bool`: value of event internal flag
evt.**set**() ➤ set event internal flag to true - unlock waiting threads
evt.**clear**() ➤ set event internal flag to False
evt.**wait**([*timeout*]) ➤ wait for event internal flag to be true - timeout is a float_delay (default to `None`=infinite blocking)

General purpose events scheduler

Module **sched** provides such a tool, adaptable to your needs ('time' unit is yours).

sc = **sched.scheduler**(*timefunc*,*delayfunc*) → `scheduler`: *timefunc* return numbers mesuring time, *delayfunc*(n) wait n time (same unit as *timefunc* output) - typically *sc* = **sched.scheduler(time.time,time.sleep)**
sc.**enterabs**(*time*,*priority*,*action*,*args*) → evtid: schedule a new event, will call *action*(*args) at *time*
sc.**enter**(*delay*,*priority*,*action*,*args*) → evtid: schedule a new event, will call *action*(*args) after *delay*
sc.**cancel**(*evtid*) ➤ remove scheduled event - may raise `RuntimeError`
sc.**empty**() → `bool`: test if scheduler events queue is empty
sc.**run**() ➤ run scheduled events at their scheduling time - see docs

Semaphores

Classes **threading**.`Semaphore` and **threading**.`BoundedSemaphore` provide simple semaphore for resources counting (without/with counter checking).

sem = **threading.Semaphore**([*value=1*]) ➤ semaphore with initial counter
sem = **threading.BoundedSemaphore**([*value*])
sem.**acquire**([*blocking*]) → `bool`/`None`: acquire the semaphore (consume one resource). *blocking* unspecified : wait & return `None` ; *blocking* true : wait & return `True` ; *blocking* false : don't wait (try) & return `True`/`False`
sem.**release**() ➤ release the semaphore (free one resource)

Condition Variables

Class **threading**.`Condition` allows threads to share state (data) protected via a `Lock`. Important : condition variables (lock) **must** be acquired when calling **wait**, **notify** or **notifyAll**. See Python docs.

cond = **threading.Condition**([*lock*]) ➤ build new condition variable, use user providen lock (`Lock` or `RLock`) else build a new `RLock`
cond.**acquire**(*args) → value: acquire cond. var. lock, return lock.acquire() value
cond.**release**() ➤ release cond. var. lock
cond.**wait**([*timeout*]) ➤ wait until notified or timeout elapsed- timeout is a float_delay (default to None=infinite blocking). Release cond. var. lock and wait for a notification/timeout then re-acquire lock.
cond.**notify**() ➤ wake up one waiting thread (if any).
cond.**notifyAll**() ➤ wake up all waiting threads.

Synchronized Queues

Module **Queue** provides a class `Queue` to store data in a synchronized FIFO queue, and two exception classes `Full` and `Empty`. In blocking mode, full queue block producers and empty queue block consumers (in non-blocking mode they raise exceptions). Other organization can be built with subclassing (see source for internal methods).

q = **queue.Queue**(*maxsize*) ➤ build new queue - infinite queue if maxsize<=0
q.**qsize**() → `int`: size of the queue - at call time
q.**empty**() → `bool`: test if queue size if 0 - at call time
q.**full**() → `bool`: test if queue size is maxsize - at call time
q.**put**(*item*[,*block*[,*timeout*]]) ➤ put *item* in queue - *block* can be true/false, *timeout* can be `None`/float_delay. May raise **Queue.Full** exception.
q.**put_nowait**(*item*) ➤ same as **put**(*item*,`False`)
q.**get**([*block*[,*timeout*]]) → item: removed from queue - *block* can be true/false, *timeout* can be `None`/float_delay - may raise **Queue.Empty** exception
q.**get_nowait**() ➤ same as **get**(`False`)

Process

Current Process

Standard module **os** has tools to get information about and manipulate current process and its environment.

Exiting

Normally Python process exit when there is no more non-daemon thread running.

sys.exit([*arg=0*]) ➤ exit via a `SystemExit` exception (may be catch) - arg is exit code

os._exit(*n*) ➤ exit without cleanup
os.abort() ➤ exit via a SIGABRT signal (signal may be handled)

Following exit codes are defined in **os** (Unix) :

EX_OK	no error
EX_USAGE	command used incorrectly
EX_DATAERR	incorrect input data
EX_NOINPUT	unavailable/inaccessible input
EX_NOUSER	unknown user
EX_NOHOST	unknown host
EX_UNAVAILABLE	required service unavailable
EX_SOFTWARE	internal error
EX_OSERR	OS error
EX_OSFILE	missing/inaccessible file
EX_CANTCREAT	can't create output
EX_IOERR	error during file I/O
EX_TEMPFAIL	temporary failure
EX_PROTOCOL	illegal/invalid/not understood protocol exchange
EX_NOPERM	not enough permissions (out of file perms)
EX_CONFIG	configuration problem
EX_NOTFOUND	missing data

You can install exit functions (for normal exit) with module **atexit**.
register(*func*[,**args*[,***kargs*]]) ➤ register function to be called with args and kargs
Registered functions are called in reverse order of registration.
Bypassed when process is terminated by a signal, an internal error, or an **os._exit**.

Environment Variables

environ → dict: environment variables - modification call **putenv** if supported
getenv(*varname*[,*default*=None]) → str: environment variable value
putenv(*varname*,*value*) ➤ set environment variable - affect later started subprocess - may cause memory leaks (see platform documentation)

Some functions also in **os.path** :
expanduser(*path*) → str: path with initial "~" or "~user" replaced
expandvars(*string*) → str: string with $name or ${name} environment variable replaced

Directory, Files, Terminal

See also Console & Interactive Input/Output (p2), and Files - Terminal Operations (p11).

chdir(*path*) ➤ change current working directory to *path*
fchdir(*fd*) ➤ change current working directory to thus represented by file descriptor
getcwd() → str: current working directory
getcwdu() → unicode: current working directory
chroot(*path*) ➤ *change process file-system root to path (Unix)*
umask(*mask*) → int: set current numeric umask and return previous one
ctermid() → str: filename of controlling terminal (Unix)
getlogin() → str: name of user logged on controlling terminal (Unix)

User, process, group IDs

pid: process id, *gid*: group id, *uid*: user id

getpid() → int: current pid
getegid() → int: effective gid (Unix)
setegid(*egid*) ➤ set process effective gid (Unix)
geteuid() → int: effective uid (Unix)
seteuid(*euid*) ➤ set process effective uid (Unix)
getgid() → int: real gid (Unix)
setgid(*gid*) ➤ set process gid (Unix)
getuid() → int: current process' uid (Unix)
setuid(*uid*) ➤ set process current uid (Unix)
setregid(*rgid*,*egid*) ➤ set process real and effective gid (Unix)
setreuid(*ruid*,*euid*) ➤ set process real and effective uid (Unix)

getpgrp() → int: current gid (Unix)
getgroups() → [int]: list of supplemental associated gid (Unix)
setgroups(*groups*) ➤ set list of supplemental associated gid (Unix)
setpgrp() ➤ call system function[1] (Unix)
getppid() → int: parent's pid (Unix)
setsid() ➤ call system function[1] (Unix)
getpgid(*pid*) → int: process group id of process id pid (0=current) (Unix)
getsid(*pid*) ➤ call system function[1] (Unix)
setpgid(*pid*,*pgrp*) ➤ set process pid group to pgrp[1] (Unix)

[1] See manual for semantics.

Timings, Priority

times() → (ut,st,cut,cst,ert): (float_delay{5}): user time, system time, children's user time, children's system time, elapsed real time
nice(*increment*) → int: renice process - return new niceness (Unix)

Memory

plock(*op*) ➤ lock program segments into memory - see <sys/lock.h> for op values (Unix)

Host Informations

strerror(*code*) → str: error message for the error code
uname() → tuple: current operating system identification, (sysname, nodename, release, version, machine) (recent Unix)

sys.byteorder → str: host native byte order big or little
sys.winver → str: version number for registry keys (Windows)
sys.platform → str: platform identifier (ex. linux2)

Following data are in **os** and in **os.path**.
defpath → str: search path for **os.exec*p***() and **os.spawn*p***() if environment PATH not defined
linesep → str: end of line char(s) for the plaftorm
devnull → str: file path of null device

Python Informations

sys.builtin_module_names → (str): names of modules compiled into interpreter
sys.copyright → str: copyright of interpreter
sys.hexversion → int: Python version with one digit by byte
sys.version → str: interpreter version + build + compiler
sys.dllhandle → int: handle of Python DLL (Windows)
sys.executable → str: name of interpreter executable binary
sys.prefix → str: directory prefix for platform independant Python files
sys.api_version → int: version of Python C API
sys.version_info → (int{3},str,int): (major, minor, micro, releaselevel, serial) - release in alpha, beta, candidate, final

Signal Handling

Standard module **signal**. See doc for general rules about signals usage in Python.
Signal handlers are callable *f*(*signalnum*,*stackframe*).

alarm(*time*) → float_delay: previous alarm remaining time - request a new SIGALRM in time seconds - cancel previous one - time≠0 (Unix)
alarm(0) → float_delay: previous alarm remaining time - cancel previous alarm (Unix)
getsignal(*signalnum*) → fct: current signal handler or SIG_IGN or SIG_DFL or None (handler not installed from Python)
pause() ➤ sleep process until a signal is received (Unix)
signal(*signalnum*,*handler*) → fct: previous handler for signal (as **getsignal**) - install new handler (maybe SIG_IGN or SIG_DFL too) - only callable in main thread

Following signal constants are defined :
SIG_DFL → 0: default signal handler function
SIG_IGN → 1: ignore signal handler function
NSIG → int: highest signal number +1

Module also defines signal numbers (Posix examples - runtime definition is platform dependant) :

SIGHUP	terminal or control processus disconnection
SIGINT	keyboard interrupt
SIGQUIT	quit request from keyboard
SIGILL	illegal instruction
SIGABRT	abort stop signal
SIGFPE	floating point error
SIGKILL	the KILL signal
SIGSEGV	invalid memory reference
SIGPIPE	pipe write without reader
SIGALRM	alarm timer elapsed
SIGTERM	termination signal
SIGUSR1	user signal 1
SIGUSR2	user signal 2
SIGCHLD	terminated/stopped child
SIGCONT	continue process (if stopped)
SIGSTOP	stop process
SIGTSTP	stop request from keyboard
SIGTTIN	read on tty while in background
SIGTTOU	write on tty while in background

… → see your platform documentation (man 7 signal on Linux).

Functions to send signals are in **os** module :
kill(*pid*,*sig*) ➤ kill process pid with signal sig (Unix)
killpg(*pgid*,*sig*) ➤ kill process group pgid with signal sig (Unix)

Simple External Process Control

Use standard module **subprocess**. It wraps external process creation and control in Popen objects. Child process exceptions raised before execution are re-raised in parent process, exceptions will have **child_traceback** attribute (string).
Note : **subprocess** tools will never call /bin/sh implicitly.

PIPE → -1: constant value used for **Popen** stdin stdout stderr params
call(**args*,***kwargs*) → int: run command with arguments, wait for completion, return retcode - convenient wrapper around Popen object

Use Popen objects as process control tools :
p =
Popen(*args*,*bufsize*=0,*executable*=None,*stdin*=None,*stdout*=None,*stderr*=None,*preexec_fn*=None,*close_fds*=False,*shell*=False,*cwd*=None,*env*=None,*universal_newlines*=False,*startupinfo*=None,*creationflags*=0)

args is a string/list of strings ["command","arg1","arg2",…]
bufsize like for **file**/**open** functions
executable can be used to provide command in place of args[0]
stdin, *stdout* and *stderr* can be **PIPE** to capture file and communicate with subprocess
preexec_fn is called just before child process execution
close_fds bool force subprocess inherited files to be closed, except 0 1 and 2
shell bool force execution of command throught the shell
cwd string specify working directory to set for subprocess start
env dictionnary specify environment variables for subprocess
universal_newlines translate all newlines to \n (like U mode for files)
startupinfo and *creationflags* are optional informations for process creation under Windows

p.**poll**() → int/None: check child process termination, return returncode attribute
p.**wait**() → int: wait for child process to terminate, return returncode attribute
p.**communicate**(*input=None*) → (stdout,stderr): send data (input string)to stdin, read data from stdout/stderr until end-of-file, wait process to terminate, return read values - data read is buffered in memory
p.**stdin** → file/None: standard input from chil process if captured
p.**stdout** → file/None: standard output from chil process if captured
p.**stderr** → file/None: error output from chil process if captured

p.**pid** → int: process ID of child process
p.**returncode** → int/None: child process return code (None if not terminated) - on Unix -N for subprocess terminated by signal N

Use **subprocess** module when possible (cleaner, simpler interface, see docs for examples). See also external module **pexpect**.

Advanced External Process Control

See following functions from **os** module.
execl(*path*,[*arg*[,…]])
execle(*path*,[*arg*[,…]],*env*)
execlp(*file*,[*arg*[,…]])
execlpe(*file*,[*arg*[,…]],*env*)
execv(*path*,*args*)
execve(*path*,*args*,*env*)
execvp(*file*,*args*)
execvpe(*file*,*args*,*env*)
With **exec**… new program replace current process (fct don't return). '**p**' versions use PATH to locate executable file. '**e**' versions use a dict *env* to setup new program environment. '**l**' versions use a positioned *arg*, '**v**' versions use list of variable *args*.
spawnl(*mode*,*path*,[*arg*[,…]])→ int
spawnle(*mode*,*path*,[*arg*[,…]],*env*) → int
spawnlp(*mode*,*file*,[*arg*[,…]])→ int
spawnlpe(*mode*,*file*,[*arg*[,…]],*env*) → int
spawnv(*mode*,*path*,*args*) → int
spawnve(*mode*,*path*,*args*,*env*) → int
spawnvp(*mode*,*file*,*args*) → int
spawnvpe(*mode*,*file*,*args*,*env*) → int
With **spawn**… new process is created. 'lpev' versions like for **exec**….
If *mode* is P_NOWAIT or P_NOWAIT0, return child pid (Unix) or process handle (Windows). If *mode* is P_WAIT, wait child termination and return its exit code (>0) or its killing signal (<0). On Windows mode can be, P_DETACH (same as P_NOWAIT but new process detached from calling process console) or P_OVERLAY (current process is replaced).
fork()→ pid: fork a child process, return 0 in child, child pid in parent (Unix)
forkpty()→ (int{2}): (pid,fd): fork using new pseudo-terminal for child - pid is 0 in child, child pid in parent - fd pseudo-terminal master end (Unix)
startfile(*path*) ➤ open file path as if double-clicked in explorer (Windows)
system(*cmd*)→ value: execute string cmd in subshell - generally return (pid/status) (Unix) or status (Windows)
wait()→ (int{2}): (pid,status) wait completion of a child process (Unix) - status=0xZZTT where ZZ=exit code, TT=signal num
waitpid(*pid*,*options*)→ (int{2}): (pid,status) (Unix):
pid>0 wait for specific process,
pid=0 wait for any child in process group,
pid=-1 wait for any child of current process,
pid<-1 wait for any process in process group -pid
option in WNOHANG, WCONTINUED, WUNTRACED
status=0xZZTT where ZZ=exit code, TT=signal num
waitpid(*pid*,*options*)→ (int{2}): (pid,status) (Windows): pid is any process handle (>0) - option ignored - status=0xZZ00 where ZZ=exit code

Status informations extraction

WCOREDUMP(*status*)→ bool: test process generated core-dump (Unix)
WIFCONTINUED(*status*)→ bool: test process continued from a job control stop (Unix)
WIFSTOPPED(*status*)→ bool: test process stopped (Unix)
WIFSIGNALED(*status*)→ bool: test exited on signal (Unix)
WIFEXITED(*status*)→ bool: test process exited via exit(2) system call (Unix)
WEXITSTATUS(*status*)→ int: if exited via exit(2), return exit parameter (Unix)

WSTOPSIG(*status*)→ int: signal having stopped process (Unix)
WTERMSIG(*status*)→ int: signal having exited process (Unix)

Pipes On Process

Three functions available in **popen2** module (and in **os** module where stdin/stdout return values are inverted).
popen2(*cmd*[,*bufsize*[,*mode*]])→ (file{2}): (stdout,stdin): execute cmd as sub-process
popen3(*cmd*[,*bufsize*[,*mode*]])→ (file{3}): (stdout,stdin,stderr): execute cmd as sub-process
popen4(*cmd*[,*bufsize*[,*mode*]])→ (file{2}): stdout_stderr,stdin): execute cmd as sub-process
Where *bufsize* is buffer size for I/O pipes, and *mode* is 'b' (binary streams) or 't' (text streams, default). Param *cmd* is a string passed to **os.system** - on Unix it can be a sequence of strings passed directly to the program without shell intervention.

On Unix, **popen2** module also defines Popen3 class (used in **popen2** and **popen3** functions) and Popen4 class (used in **popen4** function) :
Popen3(*cmd*[,*capturestderr*[,*bufsize*]])→ Popen3: *cmd*: str shell command, *captudestderr*: bool (default False)
Popen4(*cmd*[,*bufsize*])→ Popen4
Popen3 and Popen4 objects have following attributes :
p.**poll**()→ int: child return code or -1 if child not terminated
p.**wait**()→ int: child return code
p.**fromchild** → file: output from child (stdout and stderr for Popen4)
p.**tochild** → file: input to child
p.**childerr** → file: error output from child if requested else None (None for Popen4)
p.**pid** → int: child process pid
See also module **commands** (Unix).

XML Processing

Several modules to process XML are available. Some with standard SAX and DOM interfaces, others with more Pythonic interfaces. See also third party **PyXML** extension package.

SAX - Event-driven

Base functions in **xml.sax** module.
make_parser([*parser_list*]) → XMLReader: built from first parser available
parse(*filename_or_stream*,*content_handler*[,*error_handler*]) ➤ parse document using first parser available
parseString(*string*,*content_handler*[,*error_handler*]) ➤ parse string using first parser available

XMLReader Interface

Defined in **xml.sax.xmlreader**.
p = **xml.sax.make_parser**() → XMLReader object
p.**parse**(*source*) ➤ completly parse source - source is filename or URL or file-like or InputSource- input byte streams (not character streams)
p.**getContentHandler**() → ContentHandler: current one
p.**setContentHandler**(*handler*) ➤ set current content handler
p.**getDTDHandler**() → DTDHandler: current one
p.**setDTDHandler**(*handler*) ➤ set current DTD handler
p.**getEntityResolver**() → EntityResolver: current one
p.**setEntityResolver**(*handler*) ➤ set current entity resolver
p.**getErrorHandler**() → ErrorHandler: current one
p.**setErrorHandler**(*handler*) ➤ set current error handler
p.**setLocale**(*locale*) ➤ set locale for errors and warnings
p.**getFeature**(*featurename*) → current settings for feature[1]
p.**setFeature**(*featurename*,*value*) ➤ set feature to value
p.**getProperty**(*propertyname*) → current settings for property[2]
p.**setProperty**(*propertyname*,*value*) ➤ set property to value
There is also an IncrementalParser subclass interface with :
p.**feed**(*data*) ➤ process a chunk of data
p.**close**() ➤ assume end of document, check well-formedness, cleanup

p.**reset**() ➤ after close, prepare new parsing

[1] Feature names in **xml.sax.handler** as **feature_xxx**.
[2] Property names in **xml.sax.handler** as **property_xxx**.

InputSource Interface

Provide source of data for parser.
isrc.**setPublicId**(*id*) ➤ set public identifier
isrc.**getPublicId**() → unicode: public identifier
isrc.**setSystemId**(*id*) ➤ set system identifier
isrc.**getSystemId**() → unicode: system identifier
isrc.**setEncoding**(*encoding*) ➤ set encoding - must be a string acceptable for an XML encoding declaration - ignored if InputSource contains character stream
isrc.**getEncoding**() → str/None (if unknown)
isrc.**setByteStream**(*bytefile*) ➤ set input byte stream - ignored if InputSource contains character stream
isrc.**getByteStream**() → byte stream
isrc.**setCharacterStream**(*charfile*) ➤ set character (Unicode) stream
isrc.**getCharacterStream**() → character stream

Locator Interface

Instances of Locator provide these methods:
loc.**getColumnNumber**() → int: column number where current event ends
loc.**getLineNumber**() → int: line number where current event ends
loc.**getPublicId**() → str: public identifier of current event
loc.**getSystemId**() → str: system identifier of current event

Attributes Interface

Also implement parts mapping protocol (**copy**(), **get**(), **has_key**(), **items**(), **keys**(), and **values**()).
ai.**getLength**() → int: number of attributes
ai.**getNames**() → [unicode]: names of attributes
ai.**getType**(*name*)→ type of attribute name - normally 'CDATA'
ai.**getValue**(*name*)→ unicode: value of attribute name

AttributesNS Interface

Also implement Attributes interface.
ansi.**getValueByQName**(*name*)→ unicode: value of attribute qualified name
ansi.**getNameByQName**(*name*)→ (unicode{2}): (namespace, localname) for qualified name
ansi.**getQNameByName**(*namepair*)→ unicode: qualified name for (namespace, localname)
ansi.**getQNames**()→ [unicode]: qualified names of all attributes

ContentHandler Interface

Defined in **xml.sax.handler**. Its methods are handlers called when parser find XML structures.
ch = MyContentHandler() → ContentHandler subclass object
ch.**setDocumentLocator**(*locator*) ➤ set locator for origin of document events
ch.**startDocument**() ➤ beginning of document
ch.**endDocument**() ➤ end of document
ch.**startPrefixMapping**(*prefix*,*uri*) ➤ begin of a prefix-URI namespace mapping - see doc
ch.**endPrefixMapping**(*prefix*) ➤ end of a prefix-URI namespace mapping
ch.**startElement**(*name*,*attrs*) ➤ start of an element - non-namespace mode - *attrs* has an Attributes interface (may be reused - copy data)
ch.**endElement**(*name*) ➤ end of an element - non-namespace mode
ch.**startElementNS**(*name*,*qname*,*attrs*) ➤ start of an element - namespace mode - *name* is (uri,localname) - *qname* is raw XML name - attrs has an AttributesNS interface (may be reused - copy data) - *qname* may be None (upon feature_namespace_prefixes)
ch.**endElementNS**(*name*,*qname*) ➤ end of an element - namespace mode
ch.**characters**(*content*) ➤ character data - *content* is str or unicode

ch.**ignorableWhitespace**(*whitespace*) ➤ whitespaces
ch.**processingInstruction**(*target*,*data*) ➤ processing instruction
ch.**skippedEntity**(*name*) ➤ entity not processed

DTDHandler Interface

Defined in **xml.sax.handler**. Its methods are handlers called when parser need DTD relative work.
dh = MyDTDHandler() → DTDHandler subclass object
dh.**notationDecl**(*name*,*publicId*,*systemId*) ➤ notation declaration
dh.**unparsedEntityDecl**(*name*,*publicId*,*systemId*,*ndata*) ➤ unparsed entity declaration

EntityResolver Interface

Defined in **xml.sax.handler**. Its methods are handlers called when parser need external entity resolution.
er = MyEntityResolver() → EntityResolver interface object
er.**resolveEntity**(*publicId*,*systemId*) → str/InputSource: default return systemId

Exceptions

Defined in **xml.sax** module.
SAXException(*msg*[,*exception*])
  SAXParseException(*msg*,*exception*,*locator*) — invalid XML
  SAXNotRecognizedException(*msg*[,*exception*])
  SAXNotSupportedException(*msg*[,*exception*])

ErrorHandler Interface

Defined in **xml.sax.handler**. Its methods are handlers called when parser detect an error. Their *exception* parameters get SAXParseException objects.
eh = MyErrorHandler() → ErrorHandler interface object
eh.**error**(*exception*) ➤ recovererable error - parsing will continue if method return
eh.**fatalError**(*exception*) ➤ unrecoverable error - parsing must stop
eh.**warning**(*exception*) ➤ minor warning - parsing will continue if method return

SAX Utilities

Defined in **xml.sax.saxutils**.
escape(*data*[,*entities*]) → str: & < > escaped - escape other *entities* replacing mapping strings (keys) by corresponding identifiers
unescape(*data*[,*entities*]) → str: & < > unescaped - unescape other *entities* replacing mapping identifiers (keys) by corresponding strings
quoteattr(*data*[,*entities*]) → str: as **escape** + quote string to be used as attribute value
prepare_input_source(*source*[,*base*]) → InputSource: *source* is string, file-like, or InputSource - *base* is an URL string - return InputSource for parser

Class XMLGenerator is a ContentHandler writing SAX events into an XML document (ie. reproduce original document).
XMLGenerator([*out*[,*encoding*]]) → content handler: out file-like, deault to sys.stdout - encoding default to 'iso-8859-1'

Class XMLFilterBase is a default pass-throught events, can be subclassed to modify events on-fly before their processing by application handlers.
XMLFilterBase(*base*) → events filter

Features & Properties

Defined in **xml.sax.handler**. Dont give their value, but their meaning.

feature_namespaces[1] → True: perform namespace processing. False: no namespace processing (so no namespace prefixes).
feature_namespace_prefixes[1] → True: report original prefixed names and attributes used for namespace declarations.
feature_string_interning[1] → True: intern all names (elements, prefixes, attributes, namespace URIs, local names).
feature_validation[1] → True: report all validation errors.
feature_external_ges[1] → True: include all external general (text) entities.
feature_external_pes[1] → True: include all external parameter entities, including the external DTD subset.
all_features → list of all features
property_lexical_handler → optional extension handler for lexical events (like comments).
property_declaration_handler → optional extension handler for DTD-related events other than notations and unparsed entities.
property_dom_node[1] → visited DOM node (if DOM iterator) when parsing, else root DOM node.
property_xml_string → literal string source of current event (read only property).
all_properties → list of all properties names

[1] can only be read during parsing (and modified before).

DOM - In-memory Tree

Defined in **xml.dom**. Two function to register/access DOM processors, and some constants.
registerDOMImplementation(*name*,*factory*) ➤ register DOM implementation factory
getDOMImplementation([*name*[,*features*]]) → DOM implementation - name may be None - may found name in env. var PYTHON_DOM - features is [(featurename,version),...]
EMPTY_NAMESPACE → no namespace associated with a node
XML_NAMESPACE → xml prefix namespace
XMLNS_NAMESPACE → namespace URI for namespace declarations - DOM level 2 specification definition
XHTML_NAMESPACE → URI of XHTML namespace (XHTML 1.0)

DOMImplementation

impl.**hasFeature**(*feature*,*version*) → *bool*: test for supported feature in an implementation

Node

Defined in **xml.dom**, class Node is parent of XML components nodes classes.
o.**nodeType** → int: (ro) in **ELEMENT_NODE**, **ATTRIBUTE_NODE**, **TEXT_NODE**, **CDATA_SECTION_NODE**, **ENTITY_NODE**, **PROCESSING_INSTRUCTION_NODE**, **COMMENT_NODE**, **DOCUMENT_NODE**, **DOCUMENT_TYPE_NODE**, **NOTATION_NODE**
o.**parentNode** → Node/None: (ro) - None for Attr nodes
o.**attributes** → NamedNodeMap/None: attribute objects for elements, else None
o.**previousSibling** → Node/None: (ro) previous node in parent's children
o.**nextSibling** → Node/None: (ro) next node in parent's children
o.**childNodes** → [Node]: (ro) list of subnodes
o.**firstChild** → Node/None: (ro) first subnode
o.**lastChild** → Node/None: (ro) last subnode
o.**localName** → unicode/None: (ro) element name without namespace prefix
o.**prefix** → unicode/None: (ro) element namespace prefix - may be empty string or None
o.**namespaceURI** → unicode/None: (ro) URI associated to element namespace
o.**nodeName** → unicode/None: (ro) usage specified in subclasses
o.**nodeValue** → unicode/None: (ro) usage specified in subclasses
o.**hasAttributes**() → bool: test any attribute existence
o.**hasChildNodes**() → bool: test any subnode existence
o.**isSameNode**(*other*) → bool: test other refers same node
o.**appendChild**(*newChild*) → new Child: add new child node at end of subnodes - return new child
o.**insertBefore**(*newChild*,*refChild*) → new Child: add new child node before an existing subnode - at end of subnodes if refChild is None - return new child
o.**removeChild**(*oldChild*) → *oldChild*: remove a subnode, return it - when no longer used, must call *oldChild*.**unlink**()
o.**replaceChild**(*newChild*,*oldChild*) ➤ replace existing subnode with a new one
o.**normalize**() ➤ join adjacent text nodes
o.**cloneNode**(*deep*) → Node: if deep, clone subnodes too - return clone

NodeList

A sequence of nodes, usable as a Python sequence (maybe modifiable upon implementation).
o.**length** → int: number of nodes in the sequence
o.**item**(*i*) → Node/None: i^{th} item in the list

DocumentType

Subclass of Node.
o.**nodeType** → **DOCUMENT_TYPE_NODE**
o.**publicId** → unicode/None: public identifier for external subset of DTD
o.**systemId** → unicode/None: system identifier URI for external subset of DTD
o.**internalSubset** → unicode/None: complete internal subset from the document - without brackets
o.**name** → unicode/None: name of root element (as given in DOCTYPE)
o.**entities** → NamedNodeMap/None: definition of external entities
o.**notations** → NamedNodeMap/None: definition of notations

Document

Subclass of Node.
o.**nodeType** → **DOCUMENT_NODE**
o.**documentElement** → Element: root element of the document
o.**createElement**(*tagName*)→ Element: new[1] element node
o.**createElementNS**(*namespaceURI*,*tagName*)→ Element: new[1] element node with namespace - *tagName* may have prefix
o.**createTextNode**(*data*) → Element: new[1] text node containing data
o.**createComment**(*data*) → Element: new[1] comment node containing data
o.**createProcessingInstruction**(*target*,*data*) → Element: new[1] processing instruction node containing target and data
o.**createAttribute**(*name*) → Element: new[1] attribute node
o.**createAttributeNS**(*namespaceURI*,*qualifiedName*) → Element: new[1] attribute node with namespace - *tagName* may have prefix
o.**getElementsByTagName**(*tagName*) → NodeList: search for all descendants (deep search) having type *tagName*
o.**getElementsByTagNameNS**(*namespaceURI*,*localName*) → NodeList: search for all descendants (deep search) having *namespaceURI* and *localName* (part after prefix)

[1] New nodes are standalone - you must insert/associate them in/to document parts.

Element

Subclass of Node.
o.**nodeType** → **ELEMENT_NODE**
o.**tagName** → unicode: element type name - with namespace may contain colons
o.**getElementsByTagName**(*tagName*) → NodeList: search for all descendants (deep search) having type *tagName*
o.**getElementsByTagNameNS**(*namespaceURI*,*localName*) → NodeList: search for all descendants (deep search) having *namespaceURI* and *localName* (part after prefix)
o.**getAttribute**(*attname*)→ unicode: attribute value
o.**getAttributeNode**(*attrname*)→ Attr: attribute node
o.**getAttributeNS**(*namespaceURI*,*localName*)→ unicode: attribute value
o.**getAttributeNodeNS**(*namespaceURI*,*localName*)→ Attr: attribute node
o.**removeAttribute**(*attname*) ➤ remove attribute by name - ignore missing attribute
o.**removeAttributeNode**(*oldAttr*)→ Attr: remove and return *oldAttr*
o.**removeAttributeNS**(*namespaceURI*,*localName*) ➤ remove attribute

by namespace URI and name - ignore missing attribute
o.**setAttribute**(*attname*,*value*) ➤ set attribute string value
o.**setAttributeNode**(*newAttr*)→ Attr: set attribute from a new Attr node - return old one
o.**setAttributeNodeNS**(*newAttr*)→ Attr: set attribute from a new Attr node with namespace URI and local name - return old one
o.**setAttributeNS**(*namespaceURI*,*qname*,*value*)→ Attr: set attribute string *value* from a *namespaceURI* and *qname* (whole attribute name) - return old one

Attr

Subclass of Node.
o.**nodeType** → **ATTRIBUTE_NODE**
o.**name** → unicode: (ro) attribute full name - may have colons
o.**localName** → unicode: (ro) attribute name - part after colons
o.**prefix** → unicode: (ro) attribute prefix - part before colons - may be empty

NamedNodeMap

A mapping of nodes - experimentally usable as a Python mapping.
o.**length** → int: length of attributes list
o.**item**(*index*) → Attr: attribute at index - arbitrary but consistent order

Comment

Subclass of Node. Cannot have subnode.
o.**nodeType** → **COMMENT_NODE**
o.**data** → unicode: content of the comment, without <!-- and -->

Text

Subclasses of Node. Cannot have subnode. Text part in an element.
o.**nodeType** → **TEXT_NODE**
o.**data** → unicode: text content

CDATASection

Subclasses of Node. Cannot have subnode. CDATA section in a document, may have multiple CDATASection nodes for one CDATA.
o.**nodeType** → **CDATA_SECTION_NODE**
o.**data** → unicode: CDATA content

ProcessingInstruction

Subclasses of Node. Cannot have subnode. Represents a processing instruction in the XML document; this inherits from the Node interface and cannot have child nodes.
o.**nodeType** → **PROCESSING_INSTRUCTION_NODE**
o.**target** → unicode: (ro) processing instruction content up to first whitespace
o.**data** → unicode: (ro) processing instruction content after first whitespace

Exceptions

Python map DOM error codes to exceptions.

DOM codes constants	Exception
DOMSTRING_SIZE_ERR	DomstringSizeErr
HIERARCHY_REQUEST_ERR	HierarchyRequestErr
INDEX_SIZE_ERR	IndexSizeErr
INUSE_ATTRIBUTE_ERR	InuseAttributeErr
INVALID_ACCESS_ERR	InvalidAccessErr
INVALID_CHARACTER_ERR	InvalidCharacterErr
INVALID_MODIFICATION_ERR	InvalidModificationErr
INVALID_STATE_ERR	InvalidStateErr
NAMESPACE_ERR	NamespaceErr
NOT_FOUND_ERR	NotFoundErr
NOT_SUPPORTED_ERR	NotSupportedErr
NO_DATA_ALLOWED_ERR	NoDataAllowedErr
NO_MODIFICATION_ALLOWED_ERR	NoModificationAllowedErr
SYNTAX_ERR	SyntaxErr
WRONG_DOCUMENT_ERR	WrongDocumentErr

exception.**code** → int: DOM code corresponding to exception

exception.**msg** → string: message for exception

DOMException
- DomstringSizeErr — implementation limit reach
- HierarchyRequestErr — insert at wrong place
- IndexSizeErr — index range error
- InuseAttributeErr — Attr node already used in tree
- InvalidAccessErr — param/operation unsupported by object
- InvalidCharacterErr — character invalid in the context
- InvalidModificationErr — can't modify node type
- InvalidStateErr — try to use an undefined/unusable object
- NamespaceErr — change forbidden in namespace context
- NotFoundErr — node don't exist in referenced context
- NotSupportedErr — operation/type unsupported by implementation
- NoDataAllowedErr — no data for this node
- NoModificationAllowedErr — can't modify object
- SyntaxErr — invalide/illegal string
- WrongDocumentErr — impl. can't migrate nodes between docs

Databases

See Python.org wiki for a list of database interface modules.
Some interfaces are for external DB engines (MySQL, PostgreSQL, BerkeleyDB, SQLite, Metakit...), other for pure Python DB engines (gadfly, ZODB, KirkyBase, Buzhug...).

Generic access to DBM-style DBs

Standard module **anydbm** is a front-end to some available DB modules : **dbhash** (→**bsddb**→Berkeley DB), **gdbm** (→GNU dbm), **dbm** (→unix dbm) and the slow portable fallback **dumbdbm**.
Data stored in DBM-style files are accessed via a dictionary-like interface where keys and values must be str.
open(*filename*[,*flag*[,*mode*]]) → dictionary-like object: *flag* in 'r' (read-default), 'w' (write), 'c' (create if doesn't exist), 'n' (create new empty) - *mode* is unix mode flags for creation
error → tuple of exception classes from DB modules (anydbm.error,...)

Uses module **whichdb** to identify right DB module for existing file.
For new files, use first available DB module in the order of the list.
This is used by **shelve** module (see Persistence, p12).
DB modules can have specific functions related to their backend, see docs.

Standard DB API for SQL databases

Generally modules for SQL databases use the Standard Python Database API v2 (defined in PEP249).

API Informations

apilevel → str: currently '1.0' or '2.0' - '1.0' if undefined
threadsafety → int: level of thread safety

#	share module	share connections	share cursors
0	no	no	no
1	yes	no	no
2	yes	yes	no
3	yes	yes	yes

paramstyle → str: parameter marker for requests

value	params	example
'qmark'	Question mark style[1]	...WHERE name=?
'numeric'	Numeric, positional style[1 or 2]	...WHERE name=:1
'named'	Named style[2]	...WHERE name=:name
'format'	ANSI C printf format codes[1]	...WHERE name=%s
'pyformat	Python extended format codes[2]	...WHERE name=%(name)s

[1] Parameters as positional values in a sequence.
[2] Parameters as named values in a map.

Exceptions

(StandardError)
- Warning — important warning
- Error — a catch all
  - InterfaceError — problem with interface (not database)
  - DatabaseError
    - DataError — problem with data processing
    - OperationalError — problem during database operations
    - IntegrityError
    - InternalError
    - ProgrammingError — SQL programming related error
    - NotSupportedError

Exceptions classes may also be available as Connection objects attributes (optional).

Connection

connect(*dsn*[,*user*[,*password*[,*host*[,*database*]]]])→ Connection object (interface defined as a guideline) - *dsn*=data source name string
cx.**errorhandler** → fct: (optional) handler for connection errors - errorhandler(connection, cursor/None, errorclass, errorvalue) - default handler fill cx.messages and may raise exceptions
cx.**messages** → [(exception class,exception value)]: (optional) messages received from database for operations with connection
cx.**close**() ➤ terminate connection (may rollback if not commited)
cx.**commit**() ➤ commit pending transactions
cx.**rollback**() ➤ rollback pending transactions (optionnal)
cx.**cursor**()→ new Cursor object

Cursor

cu.**arraysize** → int: (RW) number of rows to fetch with **fetchmany** - default to 1
cu.**connection** → Connection: (optional) connection used by cursor
cu.**description** → [(name, type_code, display_size, internal_size, precision, scale, null_ok)]/None: describe result columns
cu.**errorhandler** → fct: (optional) handler for connection errors - errorhandler(connection, cursor, errorclass, errorvalue) - default handler fill cx.messages and may raise exceptions - inherited from connection
cu.**lastrowid** → int/None: (optional) row id of last modified column
cu.**messages** → [(*exception class*,*exception value*)]: (optional) messages received from database for operations with cursor
cu.**rowcount** → int: number of rows produced/affected by last request - -1 or None if request cant touch rows
cu.**rownumber** → int/None: (optional) 0-based index of the cursor in the result set if available
cu.**callproc**(*procname*[,*parameters*])→ (parameters) - (optional) call DB stored procedure - in result out and inout parameters may have been replaced by procedure
cu.**close**()ä close the cursor
cu.**execute**(*oper*[,*params*]) ➤ prepare and execute DB request - *params*[1] is a sequence or a mapping (see module paramstyle variable)
cu.**executemany**(*oper*,*params_seq*) ➤ like execute, with a sequence of *params* (for multiple values)
cu.**fetchone**()→ (column_value,...) / None: next row of query result, None when no more data available
cu.**fetchmany**([*size*])→ [(column_value)]: next set of rows of query result, empty list when no more data available - *size* default to *cu*.**arraysize**
cu.**fetchall**()→ [(column_value)]: all remaining rows of query result, empty list when no more data available
cu.**next**()→ (column_value) : (optional) next row of query result, raises StopIteration when no more data available
cu.**nextset**() → True/None: (optional) discards results up to next available set
cu.**scroll**(*value*[,*mode*]) ➤ (optional) - scroll cursor in current result set - mode is 'relative' (default) or 'absolute'.
cu.**setinputsizes**(*sizes*) ➤ predefine memory areas for **executeXXX** operations parameters - *sizes*=[param_size,...] - param_size=Type Object or int (max length of a string param) - param_size=None for no predefinition
cu.**setoutputsize**(*size*[,*column*]) ➤ set column buffer size for fetches of large columns (e.g. LONGs, BLOBs, etc.) by **executeXXX** - column is index in result - all columns if column not specified

cu.**__iter__()** → Cursor: (optional) object itself

> [1] Method **__getitem__** is used to get values in params, using position or name. Can use tuple or dict… or your own class objects with its **__getitem__**.
> If **next** and **__iter__** are defined, cursors are iterable.

DB types Constructors

Date(*year*,*month*,*day*) → object to hold a date value
Time(*hour*,*minute*,*second*) → object to hold a time value
Timestamp(*year*,*month*,*day*,*hour*,*minute*,*second*) → object to hold a time stamp value
DateFromTicks(*ticks*) → object to hold a date value from a given ticks value
TimeFromTicks(*ticks*) → object to hold a time value from a given ticks value
TimestampFromTicks(*ticks*) → object to hold a time stamp value from a given ticks value
Binary(*string*) → object to hold a long binary string value
SQL NULL values represented by Python None.

DB types Typecodes

STRING → string-based column (CHAR)
BINARY → long binary column (LONG, RAW, BLOBs)
NUMBER → numeric column
DATETIME → date/time column
ROWID → row ID column (CHAR)

Bulk

Tools

Batteries included: **pdb** (Python debugger), code bench with **timeit** (p10).
A must have: **pychecker**.
Take a look: **pylint**, **psyco**, **pyrex**, **pycount**, **trace2html**, **depgraph**, **coverage**, **pycover**, **Pyflakes**, **pyreverse**, **HAP**.

Links

Docs: http://www.python.org/doc/
FAQ Python: http://www.python.org/doc/faq/
PEPs: http://www.python.org/dev/peps/ (Python Enhancement Proposal)
HOWTOs: http://www.amk.ca/python/howto/
Cookbook: http://aspn.activestate.com/ASPN/Python/Cookbook/
Dive Into: http://www.diveintopython.org/

<laurent.pointal@laposte.net>
V0.67 — 2007-4-29

PQRC at http://laurent.pointal.org/python/pqrc
Long Python Quick Reference at http://rgruet.free.fr/
Original Python reference at http://www.python.org/doc